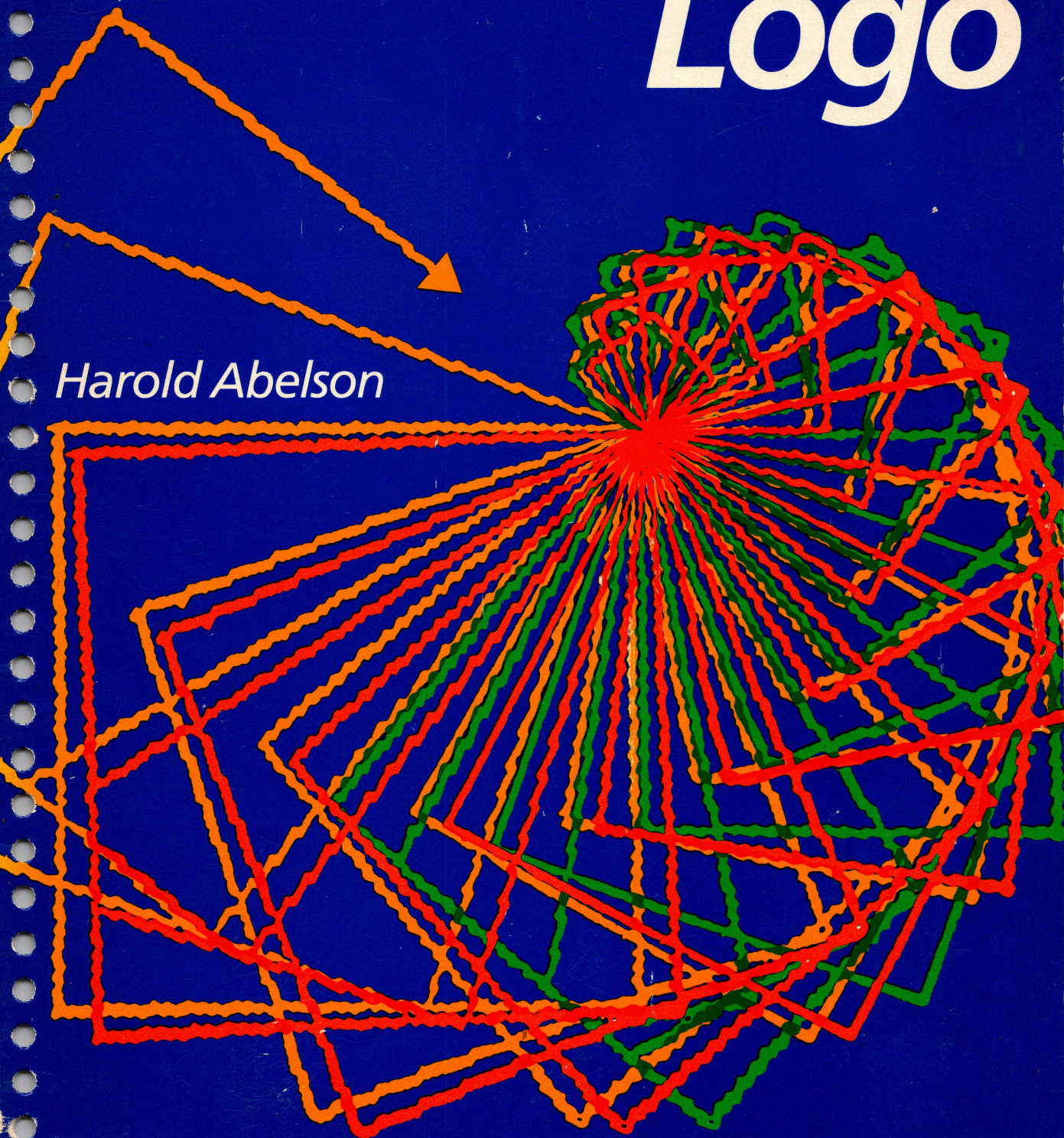


Apple[®] Logo

Harold Abelson



Apple[®] Logo

by Harold Abelson

**BYTE/McGraw-Hill
70 Main Street Peterborough, NH 03458**

Apple, Apple II, and Apple Logo are registered trademarks of Apple Computer, Inc.

Copyright © 1982 BYTE Publications, Inc. All rights reserved. No part of this book may be translated or reproduced in any form without the prior written consent of BYTE Publications, Inc.

Library of Congress Cataloging in Publication Data

Abelson, Harold
Apple Logo.

Bibliography: p.
Includes index.

1. LOGO (Computer program language)
2. Apple II (Computer)—Programming.

I. Title.

QA76.73.L63A27

1982

001.64'24

82-4545

ISBN 0-07-000425-0

AACR2

Text set in Times Roman by BYTE Publications.
Edited by Bruce Roberts and Dan Watt.
Design and Production Supervision by Ellen Klempner.
Production Editing by Peggy McCauley.
Production by Mike Lonsky.
Typeset by Donna Sweeney.
Printed and Bound by
Halliday Lithograph Corporation,
Arcata Company,
North Quincy, Massachusetts

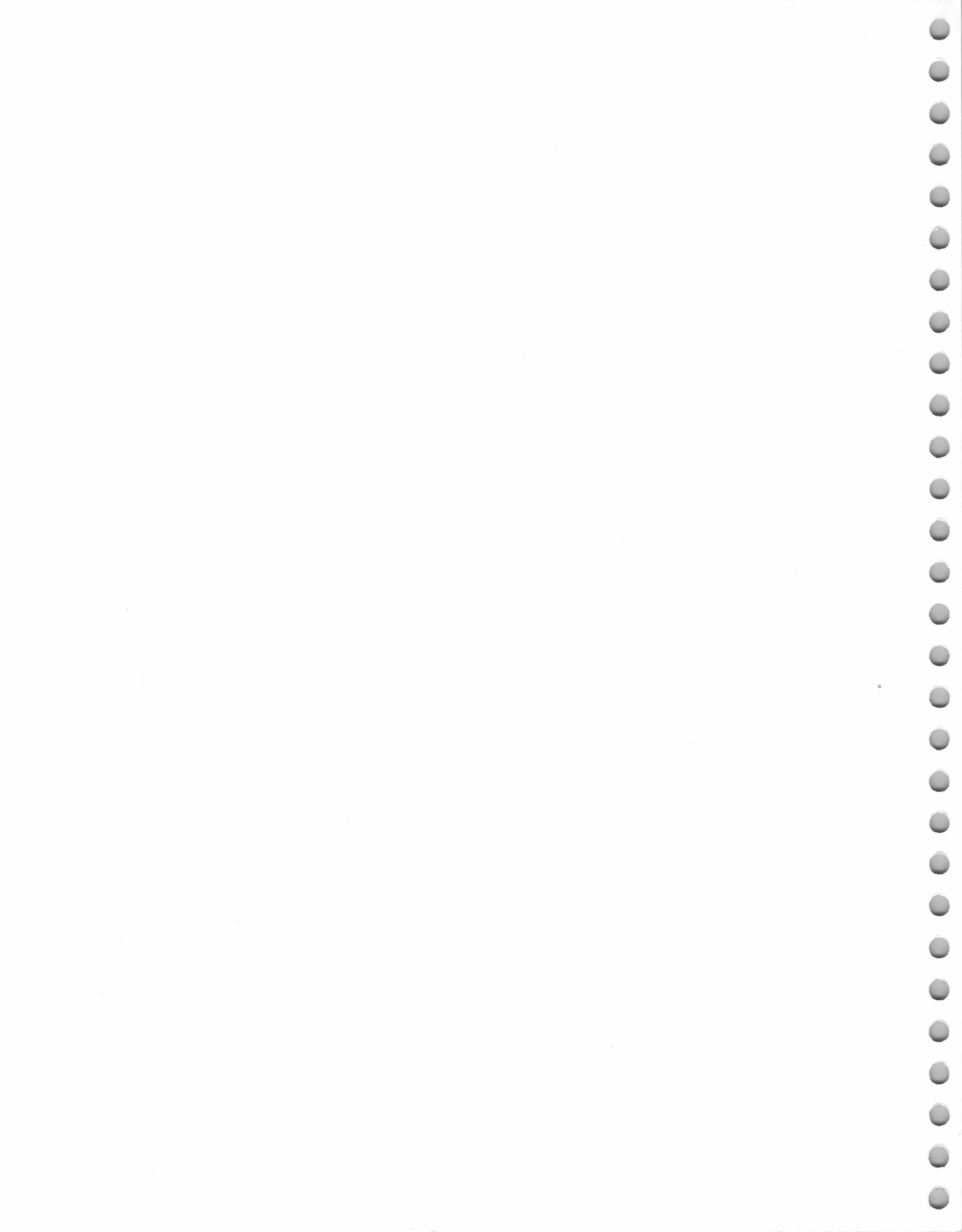
for Lynn

Table of Contents

1. A First Look at Logo	1
1.1. Before Using Logo	1
1.2. Using Logo Commands	2
1.2.1. Basic turtle commands	4
1.2.2. Correcting typing errors	5
1.2.3. Error messages	7
1.2.4. Practice with commands	7
1.3. Introduction to Procedures	10
1.3.1. Simple procedures	10
1.3.2. Defining procedures	11
1.3.3. Errors in procedures	16
1.4. Other Graphics Commands	18
1.4.1. Drawing in color	18
1.4.2. Controlling wraparound	20
1.5. Modes of Using the Screen	20
1.5.1. Nodraw mode	20
1.5.2. Definition mode	20
1.5.3. Edit mode	21
1.5.4. Draw mode	21
2. Programming with Procedures	23
2.1. Procedures with Inputs	23
2.1.1. Multiple inputs	25
2.1.2. Inputs as private names	26
2.1.3. An arc procedure	29
2.2. Repetition and Recursion	32
2.2.1. Thinking about recursion	34
2.2.2. Conditional commands and STOP	35
2.2.3. Thinking harder about recursion	37
2.2.4. Drawing trees	40
3. Projects in Turtle Geometry	45
4. Workspace, Filing, and Debugging	67
4.1. Managing Workspace	67
4.1.1. PO	67
4.1.2. ERASE	68

4.1.3. Packages	68
4.1.4. Other uses of EDIT	69
4.2. The Logo File System	69
4.2.1. Disk files	70
4.2.2. More advanced uses of the file system	71
4.3. Obtaining Hard Copy	72
5. Numbers, Words, and Lists	75
5.1. Numbers and Arithmetic	75
5.1.1. Exponential notation	76
5.1.2. Integer operations	77
5.2. Outputs	78
5.3. Words	81
5.4. Lists	83
5.5. Naming	86
5.5.1. Local and global names	88
5.5.2. Free variables	89
5.6. Conditional Expressions and Predicates	91
5.7. Details on Logo Syntax	94
5.7.1. How Logo separates lines into words	94
5.7.2. Using parentheses	95
6. Projects Using Numbers, Words, and Lists	99
6.1. Arithmetic Quiz Program	99
6.2. Random-Sentence Generators	101
6.3. Nim: a Game-Playing Program	103
6.3.1. The sub-goal plan	104
6.3.2. A simple scorekeeper	107
6.3.3. A mechanical player	109
6.3.4. Frills and modifications	112
6.3.5. A listing of the NIMPLAY procedures	113
7. Writing Interactive Programs	115
7.1. Controlling Screen Output	115
7.2. Keyboard Input	116
7.2.1. Example: instant response for very young children	117
7.2.2. Keyboard control of an ongoing process	118
7.3. Example: the Dynaturtle Program	119
7.3.1. What is a dynamic turtle?	119
7.3.2. Activities with a dynaturtle	120
7.3.3. Changing the dynaturtle's behavior	121
7.4. Input from the Paddles	123

8. Inputs, Outputs, and Recursion	125
8.1. Reversing Words and Lists	127
8.2. Recursive Procedures that Manipulate Lists	132
8.2.1. The LENGTH procedure	132
8.2.2. The PICK procedure	134
8.2.3. The MEMBER? predicate	135
8.3. Radix Conversion	137
9. Advanced Use of lists	141
9.1. Hierarchical Structures	142
9.1.1. List operations	143
9.1.2. Example: association lists	147
9.2. Programs as Data	149
9.2.1. The RUN command	149
9.2.2. The DEFINE command	152
9.2.3. The TEXT command	156
9.2.4. Adding new programming constructs	156
9.3. More Projects Using Lists	158
9.3.1. Example: the DOCTOR program	158
9.3.2. Example: the ANIMAL program	162
10. Glossary of Logo Primitive Commands	172
10.1. Graphics Commands	173
10.2. Numeric Operations	177
10.3. Word and List Operations	179
10.4. Defining and Editing Procedures	181
10.5. Conditional Expressions	183
10.6. Predicates Used with Conditional Expressions	184
10.7. Controlling Procedure Execution	185
10.8. Input and Output	185
10.9. Naming	187
10.10. Filing and Managing Workspace	188
10.11. Debugging Aids	190
10.12. Editing Commands	190
10.13. Other Control Characters	192
10.14. Primitives for Handling Properties	193
10.15. Miscellaneous and Advanced Commands	193
10.16. Error Messages	196
Appendix: TI LOGO	201
References	219
Index	221



Introduction

Logo is the name for a philosophy of education and for a continually evolving family of computer languages that aid its realization. Its learning environments articulate the principle that giving people personal control over powerful computational resources can enable them to establish intimate contact with profound ideas from science, from mathematics, and from the art of intellectual model building. Its computer languages are designed to transform computers into flexible tools to aid in learning, in playing, and in exploring.

Logo's designers are guided by the vision of an educational tool with no threshold and no ceiling. We try to make it possible for even young children to control the computer in self-directed ways, even at their very first exposure to Logo. At the same time, we believe that Logo should be a general purpose programming system of considerable power and wealth of expression. In fact, we regard these two goals as complementary rather than conflicting, since it is the very lack of expressive power of primitive languages such as BASIC that makes it difficult for beginners to write simple programs that do interesting things. More than 10 years of experience at MIT and elsewhere have demonstrated that people across the whole range of "mathematical aptitude" enjoy using Logo to create original and sophisticated programs. Logo has been successfully and productively used by preschool, elementary, junior high, high school, and college students, and by their teachers.

Some of the important features of Logo are

- Logo is a *procedural* language. Logo programs are created by combining commands into groups called procedures, and using these procedures as steps in other procedures, and so on to arbitrary levels of complexity. Each individual step of a procedure may be any primitive Logo command or any user-defined procedure. Procedures can communicate among themselves via *inputs* and *outputs*.
- Logo is an *interactive* programming language. Any Logo command, whether built into the language or defined as a procedure, can be executed by simply typing the command

at the keyboard. Logo's integrated editor makes it easy to define, execute, and modify procedures, because there is no necessity to deal with separate compilers, loaders, monitors, and so forth.

- Logo's data objects (those things that can be named by individual variables, passed directly as inputs to procedures, and returned as values) include not only numbers and character strings, but also compound structures called *lists*. Many computer languages force the programmer to manipulate data structures in terms of sequences of operations on individual numbers and character strings. In contrast, Logo's lists are functional units that can be transformed in single operations, and this makes Logo a convenient and powerful language for applications involving symbol manipulation. Moreover, the fact that Logo procedures can themselves be represented and manipulated as lists means that users can attain considerable direct control over the way that commands are interpreted, for example, to provide special interfaces to Logo for the physically handicapped or the very young.¹

Another important feature of Logo is its incorporation of a programming area called *turtle geometry*. A turtle is a computer-controlled "cybernetic animal" that lives on the display screen and responds to Logo commands that make it move (FORWARD or BACK) and rotate (LEFT or RIGHT). As the turtle moves, it leaves a trace of its path, and in this way can be used to make drawings on the display screen. For example, the following Logo procedure tells Logo how to make the turtle draw a square by repeating four times the commands "go FORWARD 100 units, turn RIGHT 90 degrees":

```
TO SQUARE
REPEAT 4 [FORWARD 100 RIGHT 90]
END
```

¹ Section 9.2.1 gives an example of a such an interface. The implementation makes use of the fact that it is possible to write Logo procedures that themselves define procedures. The book by Goldenberg [7] describes work using Logo with physically and emotionally handicapped children carried out during 1976 and 1977. More recent research in this area is discussed in the article by Weir [14].

Turtle graphics is highly successful, both as an introduction to programming for people of all ages and also as a foundation for a computer-based mathematics curriculum. In this book, we use turtle graphics to introduce the basic ideas of Logo programming, although we also cover other aspects of the language.²

Logo for microcomputers

Since its creation in 1968, Logo has been under continual development.³ As Logo is a complex and sophisticated language, most Logo work during the 1970s was conducted using large research computer systems. It is only recently that computers capable of supporting Logo have become inexpensive enough for widespread use in schools and homes. In 1979 the MIT Logo Group began an effort to adapt Logo to these machines, and this resulted in two implementations, one for the TI 99/4 home computer and one for the Apple II home computer.⁴ The examples in this book are drawn from a subsequent implementation of Logo for the Apple II home computer, developed by Logo Computer Systems, Inc., and marketed in cooperation with Apple, Inc. We hope that these implementations of Logo are but the first of many, and that the widespread availability of a simple, yet powerful, programming language will help to place the power to understand and control computers within the reach of everyone. This is precisely where that power belongs.

² In his book *Mindstorms* [12], Papert discusses the turtle as exemplary of the kind of computational "object to think with" through which technology can lead to fundamental educational change. *Mindstorms* also discusses the Logo philosophy of education and the role of computer technology in transforming education. The book by Abelson and diSessa [1] uses turtle geometry as the basis for exploring in mathematics and presents extended treatments of mathematical topics ranging from elementary geometry through General Relativity. Although turtle geometry originated as a part of the Logo language, its use is not restricted to Logo. Other languages that have incorporated turtle graphics are Smalltalk (Kay [10] and Goldberg [6]) and UCSD Pascal (Bowles [2]).

³ Logo was initially developed in 1968 as part of a National Science Foundation sponsored research project conducted at Bolt, Beranck & Newman, Inc., in Cambridge, Massachusetts (Feurzeig, *et. al.* [4]). The majority of Logo work since then has been conducted at MIT in the Artificial Intelligence Laboratory and the Division for Study and Research in Education, but there has also been significant continuing work at BBN (Feurzeig, *et. al.* [5]), at the University of Edinburgh (Howe, *et. al.* [9]), and at a number of other universities throughout the world.

⁴ The Texas Instruments Logo implementation was conducted under the sponsorship of Texas Instruments, Inc., and the project was carried out under the supervision of Seymour Papert. Principal contributors to this effort were Gary Drescher, Edward Hardebeck, Mark Gross, Leigh Klotz, and John Berlow. The Apple implementation, supported in part by the National Science Foundation, was carried out under the supervision of Harold Abelson. The implementation was accomplished by Stephen Hain, Leigh Klotz, and Patrick Sobalvarro. Both systems benefitted substantially from comments and suggestions by Andy diSessa and Dan Watt.

A guide to this book

This book is an introduction to the Logo system and to programming in Logo. You should think about learning Logo in three stages. The first stage, covered in Chapters 1 and 2, includes the basics of defining procedures and using turtle graphics to draw pictures on the display screen. Chapter 3 consists of suggestions for programming projects based on this material. Chapter 4 describes the mechanics of keeping track of procedures and saving them in disk files. The next stage in learning Logo includes writing procedures that use “data”—numbers, words, and lists as introduced in Chapter 5—and carrying out projects such as the ones presented in Chapters 6 and 7. The last section of Chapter 5 also discusses some fine points of Logo syntax, which are mostly ignored in the first four chapters. Chapters 8 and 9 cover advanced topics in Logo programming, including using recursion to deal with words and lists and using lists to represent complex data structures. Chapter 10 is a reference that describes the primitive commands included in the Logo system.

The examples included in this book draw upon research conducted over the past 10 years by members of the Logo Group at the MIT Division for Study and Research in Education and the MIT Artificial Intelligence Laboratory. Section 6.3 reprints a 1970 AI Lab Memo by Seymour Papert and Cynthia Solomon. The projects in Chapter 3 come from material prepared by Dan Watt as part of a teaching experiment conducted in the Brookline, Ma., elementary school system which is documented in [13]. I would like to thank Greg Gargarian for help in assembling this chapter. The Dynaturtle project in Chapter 7 is based on work by Andy diSessa and Dan Watt. I would also like to thank Dan Watt, Leigh Klotz, Nola Sheffer, and Richard Carter for comments on previous drafts of this book.

CHAPTER 1

A First Look at Logo

This chapter introduces the basic mechanics of using Logo. It describes how to execute simple commands and how to define and edit procedures. The examples are given in terms of using turtle graphics to draw pictures on the screen. Even though we do not, at this point, introduce more than a few commands or attempt a full explanation of the rules for writing programs, the material in this chapter and the next is sufficient to allow you to use Logo for a wide variety of interesting projects such as the ones described in Chapter 3. Try to work through this chapter at the computer keyboard, experimenting with the different features as they are introduced.

1.1. Before Using Logo

If you have never used a computer before, you will need to become accustomed to a few idiosyncrasies of computer keyboards as compared with typewriter keyboards. Be careful not to type the numeral 0 in place of the letter O, or the numeral 1 in place of the letter l. These may look alike to a person, but the keys generate different signals for the computer to interpret. Also, computer keyboards generally include a few keys not ordinarily found on typewriters. On the Apple II there is a key marked CTRL (control) that is used like an alternate shift key to type so-called *control characters*. That is to say, to type the character “control G” you hold down the CTRL key and press G (rather than trying to press both CTRL and G simultaneously). Throughout this book we specify control characters by the prefix “CTRL”, as in “CTRL-G.” Other special keys on the Apple are ESC, REPT, RESET, RETURN and the arrow keys. Also, Logo makes use of open and close bracket characters [and], which are not marked on any key, but are typed as SHIFT-N and SHIFT-M. Figure 1.1 shows a diagram of the keyboard on the Apple II together with indications of the special keys used with Logo. See the Apple Logo Technical Manual for more details on the use of the keyboard.¹ The Technical Manual also provides instructions on how to load and start the Logo system on the Apple.

¹ One important point to keep in mind when using Apple Logo is *never* to press the RESET key. This causes Logo to crash and exit to the Apple monitor. If possible, the RESET key should be disabled while using Logo.

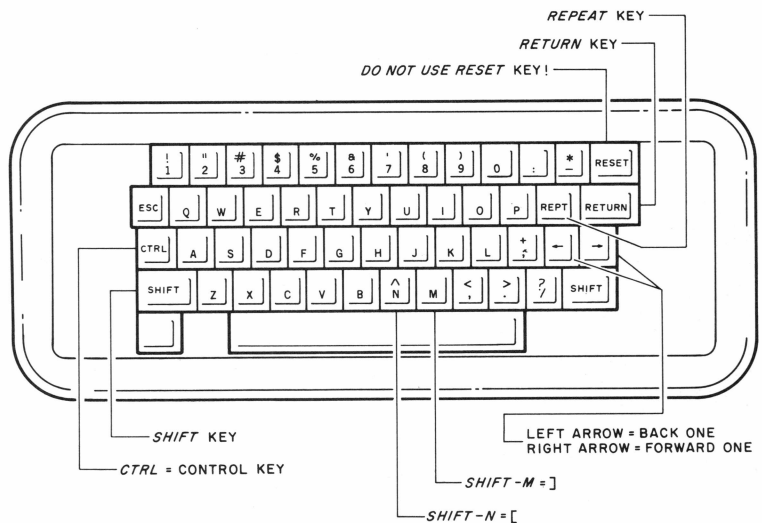


Figure 1.1: The Apple II keyboard with special keys indicated.

1.2. Using Logo Commands

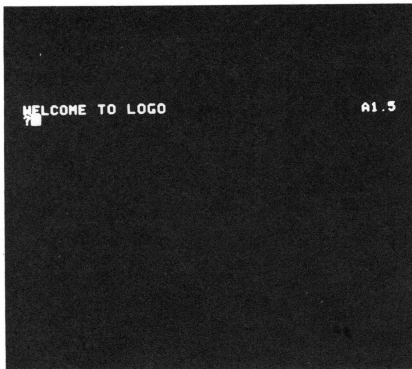


Figure 1.2: The Apple screen as it appears when Logo is first started.

Figure 1.2 is a photograph of the Apple screen as it appears when Logo is first started. The system prints a welcome message followed by a line beginning with a question mark. The question mark, called a *prompt*, indicates that Logo is waiting for you to give it a command.

To give Logo a command, you type the command and press the RETURN key. For instance, to tell Logo to print the product of 37 and 67, you type the command line

```
PRINT 37 * 67
```

that is, you type the keys P, R, I, N, T, space, 3, 7, space, *, space, 6, 7, RETURN. The computer then prints 2479, followed by a new line with a question mark prompt, indicating readiness to accept a new command. Bear in mind that when you type a command line, it is not executed until you press the RETURN key. To tell Logo to print the message "Logo is a computer language," you type the command line

```
PRINT [LOGO IS A COMPUTER LANGUAGE]
```

followed by RETURN. This example illustrates how square brackets are used in Logo to group words into *lists*. The open and closed square brackets are typed on the Apple keyboard as SHIFT-N and SHIFT-M, respectively. You can use lists in this way

to print messages on the screen, but there are many other uses for lists in Logo, and we will study these in detail in Chapters 5 and 9.

The spaces in these command lines are important, because they indicate to Logo how the line is to be broken into its component parts.² If you type the first command line omitting the space between the T and the 3 as follows:

```
PRINT37 * 67
```

then Logo will think you are telling it to execute a command named PRINT37 and complain that it does not know how to do this, by responding with the error message:

```
I DON'T KNOW HOW TO PRINT37
```

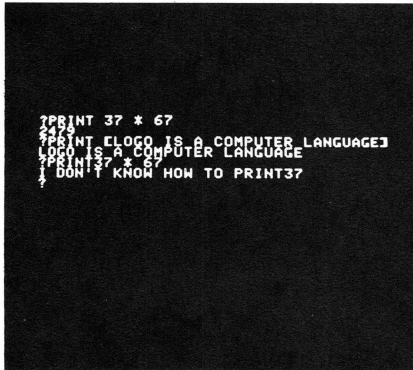


Figure 1.3: Three command lines typed to Logo, and the system's responses.

Figure 1.3 shows a photograph of the Apple screen as it appears after you give the three command lines described above, along with the computer's responses to each line. The question mark shown at the beginning of each command line is the prompt typed by Logo, and the rest of the line is the command typed by the user. In this book, when we want to emphasize the difference between the characters that you type and the characters that Logo types, we print the latter characters in *italics*. For example, the first command interaction in figure 1.3 would be printed as

```
?PRINT 37 * 67
2479
```

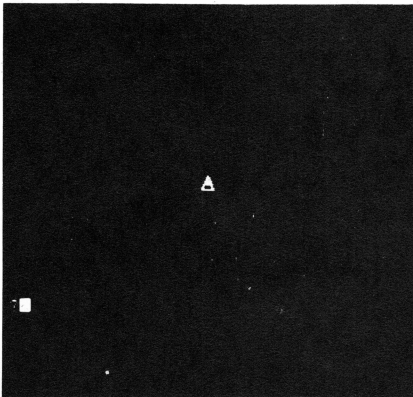


Figure 1.4: Appearance of the Apple screen when Logo enters draw mode.

In later chapters, we will see how to write Logo programs that manipulate numbers and text. But we begin our study of Logo by investigating how to use the computer to produce drawings on the display screen by issuing commands to a "creature" known as a *turtle*. To set up the screen for drawing type CLEARSCREEN and press RETURN. The screen should now appear as shown in figure 1.4, with the entire screen blank, except for a small triangle in the center and a question mark near the bottom. The question mark, as before, is the prompt indicating that Logo is ready to accept a command. When drawing, Logo reserves the four lines at the bottom of the

² Logo has some knowledge about where it is reasonable to divide lines into component parts, even when they are not separated by spaces. For example, it knows enough to interpret the string of 5 characters 37*67 as containing three elements: the number 37, the symbol *, and the number 67. However, it is a good habit to always use spaces to separate the elements of command lines, even when this is not strictly necessary. The rules that determine exactly where spaces are necessary are discussed in section 5.7.

screen for your typed commands and the computer's typed responses. The rest of the screen is for drawings. When Logo is used in this way to draw pictures on the screen, the system is said to be in *draw mode*. The original screen arrangement with no space reserved for graphics, as shown in figures 1.2 and 1.3, is called *nodraw mode*.³

1.2.1. Basic turtle commands

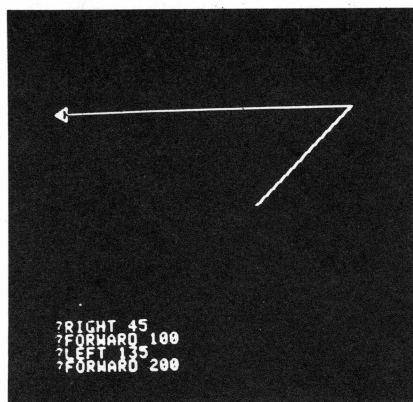


Figure 1.5: Photograph of the Apple screen showing a simple sequence of turtle commands.

```
RIGHT 45
FORWARD 100
LEFT 135
FORWARD 200
```

This should produce the wedge-shaped drawing shown in figure 1.5. Remember to terminate each command line with RETURN and to include a space between the command word and the number. If you mistype a character, you can delete the character by pressing the ← key. See section 1.2.2 for more details on correcting typing errors.

The number following the command is called an *input*. FORWARD, BACK, LEFT, and RIGHT each need one input. Logo commands may or may not require inputs, depending on the command. CLEARSCREEN is an example of a command that takes no inputs. Later on we will see examples of commands that require more than one input.

If you want to move the turtle without drawing a line, give the PENUP command. Subsequent FORWARD and BACK commands will now make the turtle move without leaving a trail. To resume drawing, give the PENDOWN command. Neither PENUP or PENDOWN takes an input. The HIDE TURTLE command causes the turtle pointer to disappear,

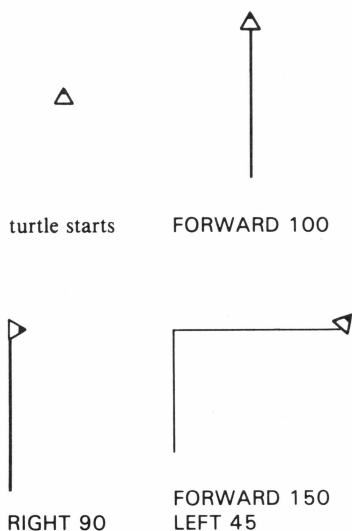


Figure 1.6: Drawing with the turtle.

³ In nodraw mode on the Apple there are 24 lines for typing. A more complete explanation of the different modes in which the Logo system operates is given in section 1.5.

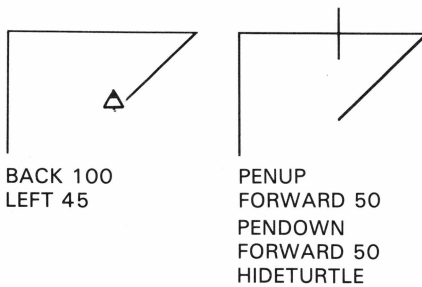


Figure 1.6 cont.

1.2.2. Correcting typing errors

although the turtle is still “there” and will draw lines if the pen is down. **SHOWTURTLE** makes the pointer reappear. Figure 1.6 illustrates the use of these commands to draw a simple picture.

If you want to start over and draw a new picture, you can use the **CLEARSCREEN** command. This erases the screen and restores the turtle to its initial location at the center of the screen, pointing straight up.

As you type Logo commands, you will undoubtedly make a few typing errors. Common errors include omitting characters, typing extra or wrong characters, and transposing characters. Logo includes a number of editing facilities that allow you to correct typing errors or otherwise modify command lines. Besides their use in editing command lines, these operations are also used in the procedure editor, which is discussed in section 1.3.2.

Inserting and deleting characters

Logo lines are edited by inserting and deleting characters. The place in the line where the change is to be made is indicated by a flashing mark called the *cursor*. When you type a character, the character is inserted into the line at the cursor position, and the cursor moves one space to the right. Therefore, if you are typing characters in the normal way, the cursor always appears at the end of the current line. The following commands allow you edit the input line:

left arrow key	Pressing the key marked ← rubs out the character immediately to the left of the cursor and moves the cursor one space to the left.
right arrow key	Pressing the key marked → moves the cursor one character to the right, <i>without</i> rubbing out any character.
REPT key	Pressing any key and holding down REPT causes the Apple keyboard to repeatedly transmit the character for as long as REPT is held down. This feature can be used to repeat any character, but it is especially useful in Logo when used in conjunction with one of the arrow keys.
CTRL-A	Pressing CTRL-A moves the cursor to the beginning of the line.

CTRL-B	Pressing CTRL-B moves the cursor one space to the left, <i>without</i> rubbing out any character.
CTRL-E	Pressing CTRL-E moves the cursor to the end of the line.
CTRL-D	Pressing CTRL-D deletes the character at the current cursor position, that is, the character over which the cursor is flashing. (D stands for "delete.")
CTRL-K	Pressing CTRL-K deletes all characters on the line to the right of the cursor. (K stands for "kill.")
CTRL-Y	Pressing CTRL-Y inserts (at the current cursor position) the characters that were killed by CTRL-K. (Y stands for "yank.")

For example, to change the line

FORWXYD 100

to

FORWARD 100

you can position the cursor under the X by pressing CTRL-B 7 times, then delete the X and the Y by pressing CTRL-D twice, and then type the characters AR. Another way to make the same change is to position the cursor under the D by pressing the CTRL-B 5 times, then delete the X and the Y by pressing ← twice, and then type AR. When the change has been made, you can execute the line in the usual way by pressing RETURN.

Recalling an input line with CTRL-Y

It is sometimes useful to be able to "recall" the input line that was just executed, either because you would like to execute the same line again, or because you would like to edit the line (to correct an error or for some other reason) and re-execute it. If you type CTRL-Y (Y stands for "yank") in response to Logo's ? prompt, your previously typed line will be retyped on the screen, with the cursor positioned at the end of the line. You can then re-execute the line by pressing RETURN, or edit the line using the editing commands previously described. Typing several CTRL-Ys will insert several copies of the previous line.

1.2.3. Error messages

If Logo cannot execute the input line, it replies with an error message. Logo's error messages attempt to be helpful in describing what went wrong. For example, if you try to execute the command line

```
PRINT 3 -
```

Logo will reply

```
NOT ENOUGH INPUTS TO -
```

because it expects to find something after the - to be subtracted from 3. Another common error message is the result of attempting to use a command that has not been defined. For instance, if you try to execute

```
TURN 100
```

Logo will respond

```
I DON'T KNOW HOW TO TURN
```

unless you have first defined a procedure named TURN.⁴ The I DON'T KNOW HOW TO error message often occurs as a result of a typing error. For example, if you type an input line like

```
FORWARD100
```

omitting the space between the D and the 1, Logo responds

```
I DON'T KNOW HOW TO FORWARD100
```

because Logo reads the entire line as a single word, which it assumes is supposed to be the name of a procedure.

When Logo responds to your command with an error message, you should try to determine the reason for the error. Sometimes it is a simple typing error. If so, you can retype the line, or, if you prefer, you can use CTRL-Y to recall and edit the line. Alternatively, the reason for the error may be hidden deep in the design of one of your programs.

1.2.4. Practice with commands

If this is your first exposure to Logo, it would be a good idea to review the material covered so far by drawing some figures

⁴ Section 1.3 explains how to define procedures.

using the turtle commands. Try to understand any error messages that occur. Following are some things to note in your exploring.

Abbreviations

Some of the commonly used Logo commands have abbreviations to help you save typing. Abbreviations for some of the commands we have seen so far are

PRINT	PR
FORWARD	FD
BACK	BK
RIGHT	RT
LEFT	LT
PENUP	PU
PENDOWN	PD
HIDETURTLE	HT
SHOWTURTLE	ST
CLEARSCREEN	CS

Multiple commands on a line

There is no restriction that each line be only a single Logo command. If you like, you can execute lines like

```
FORWARD 10 PENUP FORWARD 10
```

Logo will execute the separate commands in order, from left to right. If some command in the line causes an error, Logo will execute the commands up until the point of the error before typing an error message. However, single lines that contain many separate commands can be confusing, and it is generally better to use only one command per line.

The REPEAT command

One useful addition to your repertoire of Logo commands is REPEAT. REPEAT takes two inputs—a number and a list of commands—and repeats the commands in the list the designated number of times. For example,

```
REPEAT 4 [FORWARD 50 RIGHT 90]
```

makes the turtle draw a square. Notice that the list of commands is enclosed in square brackets.⁵ This is a very simple example of how lists are used in Logo to group things. Lists are introduced in section 5.4 REPEATs can be nested. For a pretty effect, try

Type as one line. →

```
REPEAT 10 [REPEAT 4 [FORWARD 50 RIGHT 90]
           RIGHT 36]
```

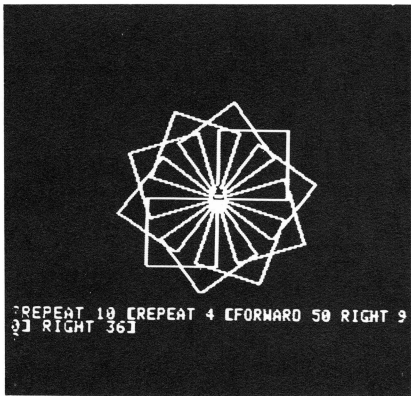


Figure 1.7: Using nested REPEATs to produce a complex drawing.

which produces the drawing shown in figure 1.7. Playing with nested REPEATs can be fun, but in terms of program clarity and power, it is much better to combine commands by defining *procedures*, as we describe in section 1.3.

Long command lines

Logo lines on the Apple screen can be at most 39 characters long. Figure 1.7 illustrates how Logo treats command lines that are longer than 39 characters. When you type the 40th character of a command line, Logo will print an exclamation point and move the cursor to the next screen line, at which point you can continue typing. To execute a long line, you type RETURN as usual. Even with this multiple line capability, no input line may be longer than 255 characters.

In this book, long command lines are not shown as they appear on the screen. Instead they are indented to make them easier to read. When you type in the program examples in the book, continue typing the indented portions as part of one long line as shown in figure 1.7.

Stopping execution with CTRL-G

When Logo is executing a command, typing CTRL-G causes it to stop whatever it is doing and wait for a new command. Logo types

STOPPED!

followed by the question mark prompt. For example, if you should start Logo executing some long process like

```
REPEAT 10000 [PRINT 1]
```

⁵ Brackets are typed on the Apple keyboard as SHIFT-N and SHIFT-M. Be sure to use square brackets [], not parentheses (), for lists.

1.3. Introduction to Procedures

1.3.1. Simple procedures



Figure 1.8a: Shape drawn by the BOX procedure.

and then think better of it, you can halt it by pressing CTRL-G. *Be sure to use CTRL-G rather than RESET in order to halt a Logo program.*⁶

You can regard Logo commands like FORWARD, PRINT, and so on, as words that the computer understands when the Logo system is started. These “built-in” words are called *primitives*. One of the most important things about the Logo language is that it makes it easy for you to teach the computer *new* words. Once you define a new word, it becomes part of the computer’s working vocabulary and can be used just as if it were a primitive. You teach Logo new words by defining them in terms of words that are already known. These definitions are called *procedures*, and this section describes the simple mechanics of how to define and edit procedures. As in the previous section, the examples are drawn from turtle graphics programs.

The following sequence of commands make the turtle draw a rectangular box as shown in figure 1.8:

```
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 20
```

You can teach the computer to execute this sequence of commands whenever you give the command BOX by defining BOX as a procedure:

```
TO BOX
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 20
END
```

- A *title line*, which consists of the word TO followed by the name you choose for the procedure.

⁶ Pressing RESET crashes the Logo system, as explained on page 1.

- A *body*, which is the sequence of command lines that make up the definition.
- The word END to indicate that this is the end of the definition.

To define this procedure, you begin by typing the command line

TO BOX

Logo responds with a special prompt character >. The prompt > in place of ? indicates that you are now in *procedure definition mode*. When you type command lines in procedure definition mode, they are not executed, but rather remembered as lines of the procedure you are currently defining. When you type END, Logo responds with a message

BOX DEFINED

to indicate that the procedure has been defined, and exits definition mode.

Once BOX is defined, it can now be used in further definitions, such as

```
TO BOXES
BOX
PENUP
FORWARD 5
LEFT 90
FORWARD 15
RIGHT 90
PENDOWN
BOX
END
```

or

```
TO PINWHEEL
REPEAT 4 [BOX]
END
```

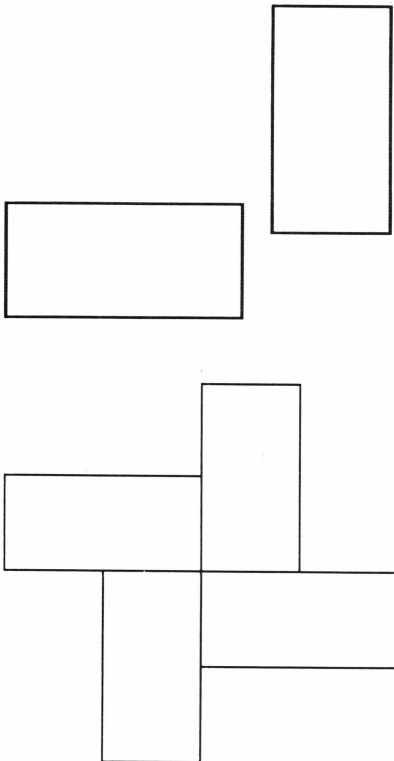


Figure 1.8b: Shapes drawn by the BOXES and PINWHEEL procedures.

which produce the drawings shown in figure 1.8. When a procedure is used as part of the definition of a new procedure, it is referred to as a *subprocedure* of the new procedure.

Remember that once a procedure is defined, you can consider it to be just another word that the computer “knows.” You tell Logo to execute any of these procedures in the same way that you tell it to execute a primitive command—by typing the name of the command followed by RETURN.

1.3.2. Defining procedures using the Logo Screen Editor

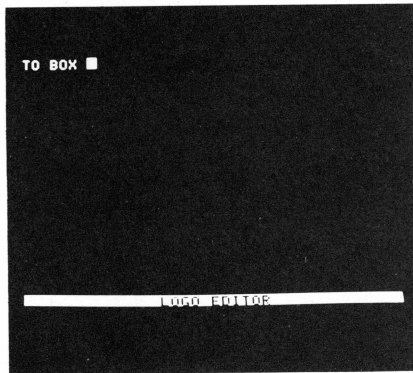


Figure 1.9: The Apple screen as it appears when you enter edit mode by typing EDIT "BOX.

A more convenient way to define procedures is to use the Logo *screen editor*. The following paragraphs describe how to use the editor to define procedures such as the BOX procedure shown above. When Logo gives its question mark prompt, you type

EDIT "BOX

and press RETURN. The screen should now be clear, except for your typed line TO BOX at the top, and a “reverse video” line at the bottom of the screen that reads LOGO EDITOR to indicate that you are now using the procedure editor, or, are in so-called edit mode. This configuration is shown in figure 1.9.

The procedure editor

In edit mode, you type in the procedure definition, line by line. The major difference between typing in the procedure editor and typing regular Logo commands is that pressing RETURN merely moves the cursor to the beginning of the next line, rather than telling Logo to execute the current line as a command. Logo is now *storing* your command lines as part of the procedure, rather than executing them. All of the editing commands described in section 1.2.2 work in the usual way, except for the following differences which occur when the cursor is at the beginning or end of a line:

- Pressing the right arrow key when the cursor is at the end of a line moves the cursor to the beginning of the next line.
- Pressing the CTRL-B when the cursor is at the beginning of a line moves the cursor to the end of the previous line.
- Pressing left arrow when the cursor is at the beginning of a line combines that line with the previous line.
- Pressing CTRL-D when the cursor is at the end of a line combines that line with the following line. You can view the effect of left arrow and CTRL-D as one of deleting the RETURN that separates the two lines.

Besides the regular line-editing commands, you can also use

- CTRL-N Pressing CTRL-N moves the cursor *down* one line. (N stands for “next.”)
- CTRL-O Pressing CTRL-O *opens* a new line at the position of the cursor. That is, it is equivalent to typing RETURN and then restoring the cursor to its original position.
- CTRL-P Pressing CTRL-P moves the cursor *up* one line. (P stands for “previous.”)

Exiting the editor

After you have typed in the procedure definition and have edited the text to your satisfaction, you press CTRL-C. The definition is now be processed. Logo then types the message

BOX DEFINED

(with BOX replaced by whatever name you actually used in the title line) and prompts for a new command. Logo always returns from edit mode to nodraw mode (page 4). To restore the turtle, you give the SHOWTURTLE command.⁷

Aborting an edit with CTRL-G

Typing CTRL-C exits the editor and installs your procedure definition. Sometimes, however, you may enter edit mode and then change your mind about performing an edit. If you type CTRL-G while in edit mode, Logo will return to nodraw mode, but any editing you have just done will be ignored.

Changing procedure definitions

Suppose you want to change the definition of a procedure. For example, you may want to change the definition of PINWHEEL on page 17 from

```
TO PINWHEEL
REPEAT 4 [BOX]
END
```

to

⁷ The Apple Logo implementation uses the same region of computer memory to store the screen image as it does to store the text being edited. Therefore, using the editor wipes out any graphics image you have on the screen.

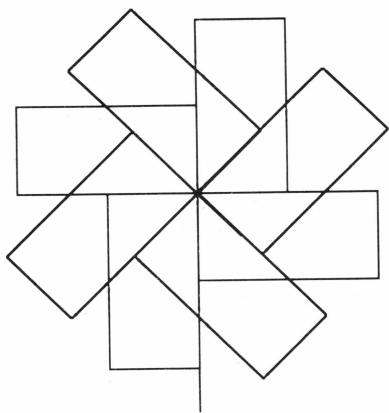


Figure 1.10a: Shape drawn by the modified PINWHEEL procedure.

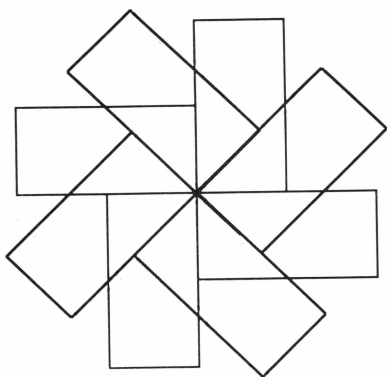


Figure 1.10b: Shape drawn by the FAN procedure.

```
TO PINWHEEL
REPEAT 8 [RIGHT 45 BOX]
RIGHT 180
FORWARD 50
END
```

so that it now makes the drawing shown in figure 1.10. To accomplish this, you give the command

```
EDIT "PINWHEEL
```

Logo now places you in edit mode with the original text of the PINWHEEL procedure shown on the screen. Now edit the definition, inserting and deleting text using any of the editing commands described above. When you have finished editing, you type CTRL-C. Logo prints

```
PINWHEEL DEFINED
```

and prompts for a new command as before.

When you change a procedure definition, the computer then uses the new, not the old, definition any time the procedure is executed.

Changing the procedure's name (by editing the title line) is equivalent to defining a new procedure with the new title. For example, if you edit the PINWHEEL definition to read

```
TO FAN
REPEAT 8 [RIGHT 45 BOX]
END
```

Logo will type

```
FAN DEFINED
```

when you exit edit mode, and the computer will remember *both* FAN and PINWHEEL.

Printing procedures and titles

In order to see the definition of a procedure, you can use the PO command (PO stands for "printout") followed by the name of the procedure, with quotation marks before it. Here is an example:

```

PO "PINWHEEL
TO PINWHEEL
REPEAT 8 [RIGHT 45 BOX]
RIGHT 180
FORWARD 50
END

```

Another useful Logo command is POTS (for “printout titles”), which lists the title lines of all procedure that are currently defined, for example:

```

POTS
TO PINWHEEL
TO FAN
TO BOXES
TO BOX

```

See section 4.1.1 for more details on PO.

Defining more than one procedure at a time

If you like, you can use the editor to define more than one procedure definition at a time. You simply type in the definitions in sequence. Be sure to end each separate procedure definition with END.⁸

As with multiple commands on a line, defining more than one procedure at once can cause confusion, because if some procedure definition is badly formed and causes a definition error (see section 1.3.3 below), the procedure definitions that follow it will not be processed. It is generally better to enter and exit edit mode for each procedure separately.

More than one screenful

Sometimes, either because you have a very long procedure definition, or, more commonly, because you are defining many procedures at once, you may want to edit more lines of text than can fit on the screen at once. Logo allows you to do this. If the cursor is at the bottom of the screen and you press RETURN, the lines of text will scroll upwards to produce a new blank line. In general, the screen can be thought of as a window onto a much longer page of text that scrolls up and down so that the part you are editing is always within the window.

⁸ Actually, you do not have to type END at the end of the final definition. So if you define only one procedure at a time, you never have to type END at all. When you press CTRL-C, Logo types its {xxx} DEFINED message for each procedure in turn.

Additional editing commands to help you deal with large amounts of text are

- CTRL-V Pressing CTRL-V scrolls the text forward one screenful.
- ESC-V Pressing ESC and then V scrolls the text back one screenful.
- CTRL-L Pressing CTRL-L scrolls the text so that the line containing the cursor is approximately in the center of the screen.

Long lines in procedures

As is the case with command lines, lines in Logo procedures can be more than one screen-line (40 characters) long, up to 255 characters. When you type a long line using the procedure editor, Logo prints an exclamation point when you reach the right edge of the screen and continues with the cursor at the left of the next screen line. Note that the exclamation point at the right of the screen is, as far as the editing commands are concerned, not really part of the input line, but rather a visual indicator that the input line continues on to the next screen line.

1.3.3. Errors in procedures

If Logo encounters an error while executing a procedure, it prints an error message as described in section 1.2.3 together with three pieces of information:

- A description of the error.
- The name of the procedure in which the error occurred.
- The line that contained the error.

For example, suppose you define the procedure

```
TO BLOCK
ELL
RIGHT 90
ELL
END
```

and the definition of the subprocedure ELL contains a typing error (in the first line of the procedure):

```

TO ELL
FORWAXD 50
RIGHT 90
FORWARD 25
END

```

Then if you give the command BLOCK, Logo will run until it tries to execute the first line in ELL for the first time at which point it will type

```

I DON'T KNOW HOW TO FORWAXD IN ELL:
FORWAXD 50

```

At this point you should edit ELL and correct the mistyped line.

Errors in procedure definitions

When Logo processes a procedure definition, it does not look for errors in the lines that make up the body of the definition. For example, if you make a typing error, as in the second line below:

```

TO PINWHEEL
REPEAT 8 [RIGXT 45 BOX]
END

```

the fact that RIGHT has been mistyped as RIGXT will cause an error when Logo attempts to *execute* PINWHEEL, not when you define the procedure.⁹ On the other hand, there are certain things that can cause errors when you type CTRL-C and the definition is processed. For example, you may mistakenly use the editing operations to remove the word TO from the title line while you are editing, or cause the definition to be badly formed in some other way.¹⁰ In such a case, Logo will print an

⁹ One very good reason for this is that it is always possible that you *did* mean to type RIGXT, and you will be defining a procedure named RIGXT before using PINWHEEL. One facility that a computer language can provide to encourage sound programming practices is to make it possible to write definitions in terms of procedures that have not yet been defined.

¹⁰ Logo will complain, for example, if you try to define a procedure with the same name as some Logo primitive. For example, if you attempt to define a procedure named FORWARD, Logo will respond with the error message

```
FORWARD IS A PRIMITIVE
```

In fact, there are facilities built into Apple Logo which allow you to redefine primitives. See the technical manual for details.

error message instead of the {xxx} *DEFINED* message. If you wish, you can then reenter the editor and attempt to salvage the definition. To do this you give the Logo command *EDIT* without specifying an input. This places you back in edit mode with the last text you were editing shown in the screen.¹¹

1.4. Other Graphics Commands

Section 10.1 gives a complete list of the graphics commands that are built in to Logo. This section describes some of the ones that we have not already seen.

In addition to the turtle commands *FORWARD*, *BACK*, *LEFT*, and *RIGHT*, Logo allows you to move the turtle by specifying x,y Cartesian coordinates. The *SETPOS* command takes a list of two numbers as input and moves the turtle to the corresponding x,y screen location. There are also commands *XCOR* and *YCOR* that output the turtle's position.¹² The *SETHEADING* command rotates the turtle so that it faces in a specified direction, and the *TOWARDS* command outputs the direction in which the turtle should be pointing in order to face a specified point. Giving the command *HOME* moves the turtle back to its initial position at the center of the screen and facing straight up. *CLEAN* erases any drawings on the screen without changing the turtle's position.

1.4.1. Drawing in color

If you have a color TV monitor, you can use the *SETPC* command ("set pencolor") to change the color of the lines that the turtle draws. You can also use the *SETBG* command ("setbackground") to make the turtle draw on backgrounds of various colors. Both *SETPC* and *SETBG* take a color number as input. The correspondence of colors to numbers is as follows:¹³

black	0
white	1
green	2
violet	3
orange	4
blue	5

¹¹ For the reason described in note 7, you cannot make use of this reentry feature if you have used graphics in the meantime.

¹² See section 5.2 on how to use outputs.

¹³ The actual color that appears on the screen corresponding to any of these color names can vary greatly depending on the adjustment of the TV monitor. Also, if you have a black-and-white monitor, the "colors" will appear as grey, so you can use *SETPC* to draw pictures in black, white, and grey.

You can also use the **PENREVERSE** command (abbreviated **PX**). This will cause the turtle to “reverse” the color of all dots that it passes over. The actual color produced depends on both the background color and the color of the dot the turtle is passing over, but in all cases, reversing the color of a dot and then reversing it again restores the original color. The turtle also carries an “eraser” that can be used to erase lines on the screen. The **PENERASE** command (abbreviated **PE**) lowers the eraser. Giving the **PENUP** or **PENDOWN** commands lifts the eraser.

If you do not explicitly give any background or pencolor commands, Logo will default to background 0 and pencolor 1.

Drawing on colored backgrounds

When drawing on a colored background (2 through 5), only two of the four colors—green, violet, blue, orange—are available. When the background is green or violet, blue and orange cannot be used: pencolor 4 draws in green and pencolor 5 draws in violet. When the background is blue or orange, pencolor 2 draws in orange and pencolor 3 draws in blue. Also, if you draw a picture on the screen and then change the background color, the colors of the lines in the picture may change, or other strange effects may result.¹⁴ Some people like to experiment with these effects, whereas others find them annoying. If you don’t like them, you should make a habit of clearing the screen before changing the background color.

Drawing without color control

In order to obtain clear colors with the Apple computer, the Logo system must draw thicker lines than would otherwise be necessary. This means that drawings do not look as precise as they would if only thin lines were drawn. If you don’t care about color, you will obtain better-looking drawings by using thin lines. To do this, select background 6 and pencolor 1.

Summary chart

The restrictions in the paragraphs mentioned below are summarized in the following table, which shows the actual effect of each pencolor number drawn with each background number.

¹⁴ This unfortunate situation is the result of compromising with a limitation of the Apple computer color system, which does not allow, for example, green dots to appear very close to orange dots.

BACKGROUND

P	O	1	2	3	4	5	6
E 0	black	black	black	black	black	black	black
N 1	white	white	white	white	white	white	white
C 2	green	green	green	green	orange	orange	white
O 3	violet	violet	violet	violet	blue	blue	white
L 4	orange	orange	green	green	orange	orange	white
O 5	blue	blue	violet	violet	blue	blue	white
R							

1.4.2. Controlling wraparound

The turtle screen is 280 “turtle steps” wide by 240 steps high. If you give a command that moves the turtle outside this range, the turtle wraps around to appear at the opposite edge of the screen. That is to say, driving the turtle off the top of the screen makes the turtle reappear at the bottom of the screen to continue drawing. Driving the turtle off the right edge of the screen makes it reappear at the left of the screen, and so on. As alternatives to this wraparound behavior, Logo can be made to act in two other ways in response to moving the turtle beyond the screen boundaries. The first way is to arrange things so that attempting to drive the turtle off the edge of the screen produces a TURTLE OUT OF BOUNDS error message. This is done by issuing the Logo command FENCE. As a second alternative, you can use the WINDOW command, which makes the screen act like a “window” into a much larger turtle field. In this case, the turtle can move beyond the screen borders, but you can see only that portion of the turtle field that lies within the window.¹⁵ The WRAP command restores Logo’s original wraparound behavior.

1.5. Modes of Using the Screen

This chapter has presented the basics of executing Logo commands and defining simple procedures. As a summary, we note that Logo uses the display screen in three different ways, or “modes.”

1.5.1. Nodraw mode

Logo starts in nodraw mode. You type in command lines, terminated with RETURN. Logo executes the line and prints a response, if appropriate.

1.5.2. Definition mode

Executing the TO command places Logo in procedure definition mode. When you type in commands lines, Logo does

¹⁵ The size of the large field is 40960 steps high by 32768 steps wide. Attempts to move outside this range will cause a TURTLE OUT OF BOUNDS error. (The precise screen bounds given here depend on how the Logo graphics “aspect ratio” has been set. See the description of SETSCRUNCH in Chapter 10.)

not execute them, but rather stores these as part of the procedure being defined. The special prompt `>`, in place of the usual `?`, indicates that Logo is in definition mode. Typing `END` causes Logo to leave this mode.

1.5.3. Edit mode

Executing the `EDIT` command places Logo in edit mode, which allows you to use the procedure editor as described in section 1.3.2. Typing `CTRL-C` exits edit mode, processes the definitions, and returns to nodraw mode. Typing `CTRL-G` exits edit mode and enters nodraw mode without processing the definitions.

1.5.4. Draw mode

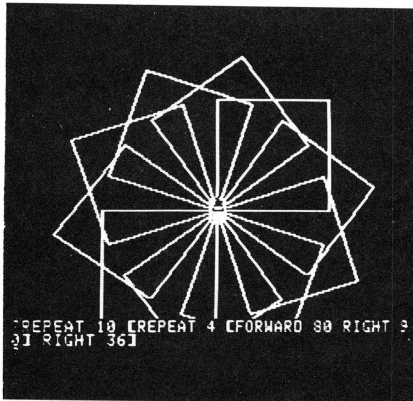


Figure 1.11: In splitscreen mode, a drawing may be partially hidden by text.

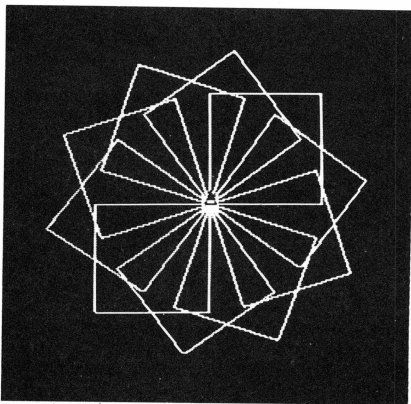


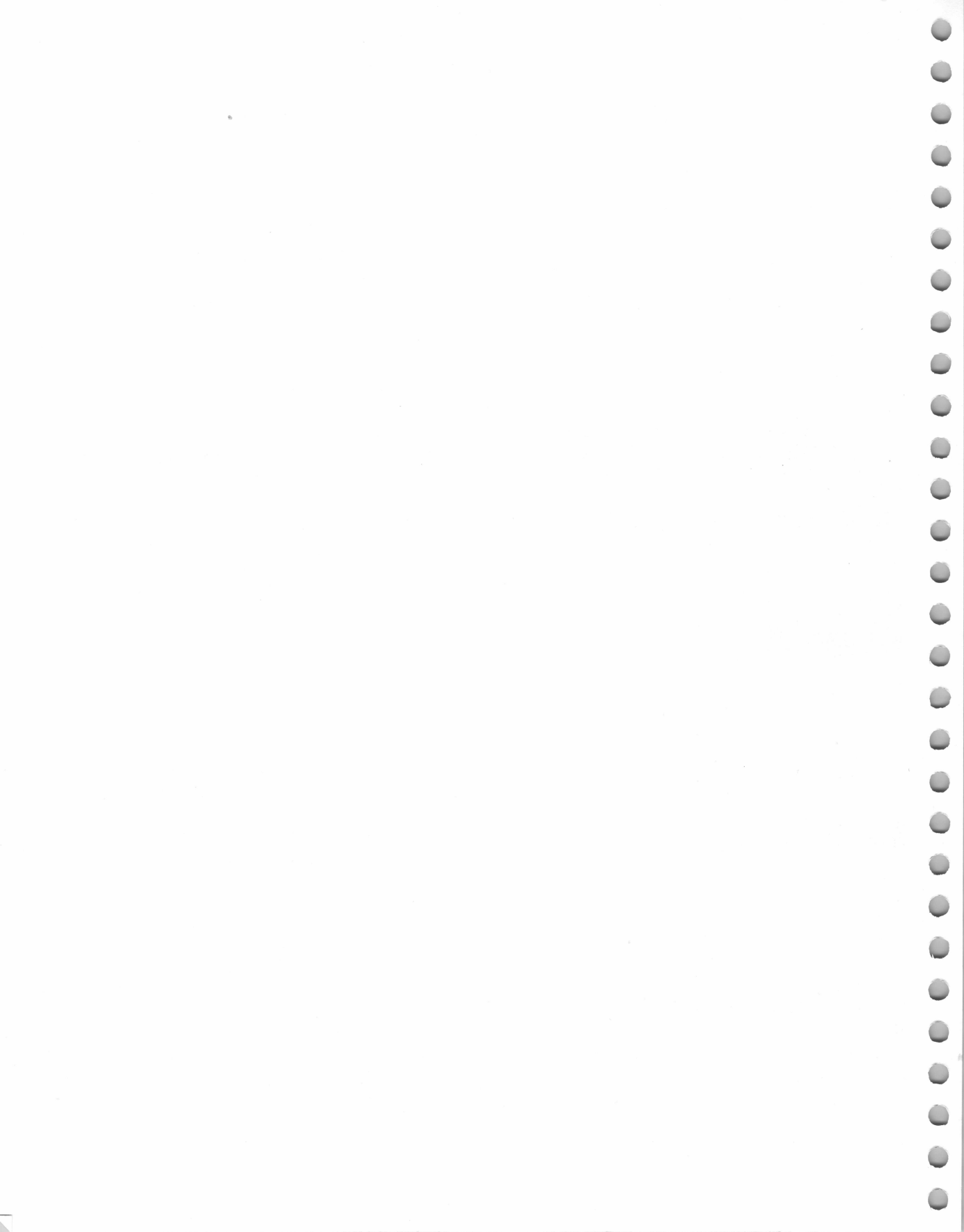
Figure 1.12: The drawing in figure 1.11, shown in fullscreen mode.

Executing any graphics command causes Logo to enter draw mode as shown in figure 1.4, with the screen cleared and the turtle at the center. In draw mode, you use the turtle for drawing on the screen. The `TEXTSCREEN` command exits draw mode and enters nodraw mode. Actually, in Apple Logo, there are two different types of draw mode.

Splitscreen mode is the normal way in which draw mode is used. Four lines at the bottom of the screen are reserved for text, and the rest of the screen shows the field in which the turtle moves. The turtle field actually extends to the bottom of the screen and so is partially masked by the four-line text region. In *fullscreen mode* the text region disappears, and you can see the entire turtle field. You can still type commands, but your typing will not be visible. If Logo needs to type an error message, it will first enter splitscreen mode so that the message will be visible. Figure 1.11 shows a turtle drawing partially hidden by the text in splitscreen mode, and figure 1.12 shows the results of switching to fullscreen mode.

You can use the characters `CTRL-L` and `CTRL-S` to switch back and forth between splitscreen and fullscreen modes. Pressing `CTRL-L` while in splitscreen mode enters fullscreen mode. Pressing `CTRL-S` restores splitscreen mode. It is also sometimes convenient to be able to switch back and forth under program control. This can be done by using the commands `SPLITSCREEN` and `FULLSCREEN`.

It is convenient to be able to temporarily view more than four lines of typing without destroying the picture being drawn. To switch back to nodraw mode, type `CTRL-T` (or give the `TEXTSCREEN` command). You can then return to draw mode with `CTRL-L` or `CTRL-S` (or `FULLSCREEN` or `SPLITSCREEN`) or by issuing any graphics command. This feature is especially useful if you are drawing and do something that causes Logo to type an error message that is more than four lines long. If so, you can type `CTRL-T` to read the message and then type `CTRL-S` to return to your drawing.



CHAPTER 2

Programming with Procedures

In the introduction we stressed that the ability to define procedures is one of the powerful features of the Logo language. In this chapter we explain more about how procedures can be used and, in particular, how they can be used to build up complex programs in simple steps. With the material covered in this chapter, you should have enough information about Logo to undertake many projects in turtle geometry such as the ones presented in Chapter 3.

2.1. Procedures with Inputs

The procedures discussed in section 1.3 do exactly the same thing each time they are executed. Each turtle procedure draws the same drawing each time. Contrast this with a command like FORWARD.

```
FORWARD 100
```

does not draw exactly the same thing as

```
FORWARD 50
```

The fact that the FORWARD command takes an *input* is what enables you to use this one command to draw lines of all different lengths.

In Logo, you can define procedures that take inputs. Consider, for example, the following procedure, which draws a square 100 units in a side:

```
TO SQUARE
REPEAT 4 [FORWARD 100 RIGHT 90]
END
```

Whenever you give the command SQUARE, the turtle draws a square with side 100. You can change the definition of SQUARE so that it can be used to draw squares of all different sizes:

```
TO SQUARE :SIDE
REPEAT 4 [FORWARD :SIDE RIGHT 90]
END
```

The new SQUARE procedure takes an input that specifies the side of the square to be drawn. The procedure is executed just like any Logo command that takes an input; that is, to draw a square of side 100 you type

SQUARE 100

To draw a square of side 50 you type

SQUARE 50

and so on.¹

The definition of SQUARE illustrates the general rule for defining procedures that take inputs. You choose a name for the input and include it in the procedure title line, preceded by a colon.² Now you use the input name (with the colon) wherever you would normally use the value of the input in the procedure body.

Here's another example. You can modify the original (side 100) SQUARE procedure to draw the diagonals of the square, using the fact that the length of the diagonal is the square root of 2 (about 1.41) times the length of the side.

```
TO DIAG
REPEAT 4 [FORWARD 100 RIGHT 90]
RIGHT 45
FORWARD 141
LEFT 45
BACK 100
LEFT 45
FORWARD 141
END
```

¹ A common beginner's mistake is to type SQUARE :100, based on the (reasonable) misunderstanding that the colon means something like "here is your input." Instead, as we shall see below, the colon as used in :SIDE means "the value associated with the name SIDE."

² Logo tradition is to pronounce the colon as "dots." That is, :SIDE is pronounced "dots SIDE."

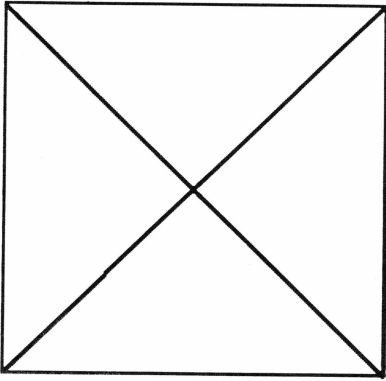


Figure 2.1: Shape drawn by the DIAG procedure.

Figure 2.1 shows the shape drawn by this procedure. To draw the shape in all different sizes, you can use

```
TO DIAG :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
RIGHT 45
FORWARD :SIZE * 1.41
LEFT 45
BACK :SIZE
LEFT 45
FORWARD :SIZE * 1.41
END
```

2.1.1. Multiple inputs

Logo procedures may be defined to accept more than one input. You simply choose a name for each input and include it in the title line, preceded by a colon. For example, the following two-input procedure can be used to draw rectangles of varying sizes and shapes:

```
TO RECTANGLE :HEIGHT :LENGTH
FORWARD :HEIGHT
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :HEIGHT
RIGHT 90
FORWARD :LENGTH
RIGHT 90
END
```

As shown in figure 2.2, executing the command

```
RECTANGLE 100 10
```

draws a long, skinny rectangle, whereas

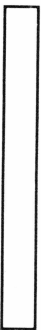


Figure 2.2a: Rectangle drawn by RECTANGLE 100 10.

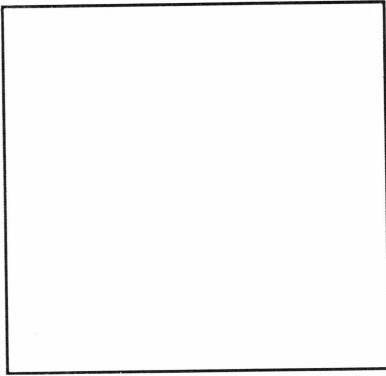


Figure 2.2b: Rectangle drawn by RECTANGLE 100 100.

2.1.2. Inputs as private names

RECTANGLE 100 100

draws a square.

Defining a Logo procedure involves grouping together a series of commands under a *name* chosen by the programmer. Using inputs also involves naming, but in a different sense. Although a new procedure is incorporated as part of Logo's working vocabulary, the name of an input is *private* to the procedure that uses the input.

Since input names are private, different procedures may use the same names for inputs without these names interfering with each other.³ One way to think about this is to imagine that each time a procedure is executed, it sets up a "private library" that associates to its input names the actual input values with which the procedure was called. When the procedure executes a line that contains an input name (signaled by :) it looks up the value in the library and substitutes the value for the name. For example, the previous RECTANGLE procedure, called with

RECTANGLE 10 100

would set up a private library as shown in figure 2.3.

RECTANGLE	
HEIGHT	10
LENGTH	100

Figure 2.3: Private library set up by executing RECTANGLE 10 100.

The input values are associated with the input names in the order in which they appear in the title line. In this case, the first input, 10, is associated with the first input name, HEIGHT, and the second input, 100, with the second name, LENGTH.

We've already seen in Chapter 1 that the individual steps in a procedure can themselves be procedures. Since each procedure maintains its own private library of input values, there is no

³ In computer science terminology, inputs are said to be "local variables" to the procedure in which they are defined.

conflict between the input names used by the different procedures. For example, here is RECTANGLE used as part of a procedure for drawing a flag, shown in figure 2.4:

```
TO FLAG :HEIGHT
FORWARD :HEIGHT
RECTANGLE (:HEIGHT / 2) :HEIGHT
BACK :HEIGHT
END
```

The FLAG procedure draws a “pole” of a specified HEIGHT, then draws on top of the pole a rectangle of dimensions HEIGHT / 2 by HEIGHT, then moves the turtle back to the base of the pole. Note the use of parentheses around (:HEIGHT / 2). These are not actually necessary for Logo to understand what is meant, but they make the program easier to read.⁴

Let’s examine in detail what happens when you execute the command

FLAG 50

This creates a private library for FLAG in which HEIGHT is associated to 50, and begins executing the definition of FLAG, starting with the first line

FORWARD :HEIGHT

Looking in the private library, Logo finds that 50 is the value associated with HEIGHT, so it makes the turtle go FORWARD 50. Next it must execute the line

RECTANGLE (:HEIGHT / 2) :HEIGHT

To do this, Logo first determines the values of the two inputs that must be given to RECTANGLE. The first input is half the value of HEIGHT, or 25, and the second input is HEIGHT itself, or 50. Now RECTANGLE is called with inputs 25 and 50. This sets up a private library for RECTANGLE in which the names of RECTANGLE’s inputs, HEIGHT and LENGTH, are associated to 25 and 50, respectively. The entire picture is as shown in Figure 2.5. Even though the name HEIGHT is associated with 50 in FLAG’s library and with 25 in RECTANGLE’s library there is no conflict between the two. Each procedure looks up its own values in its own library.

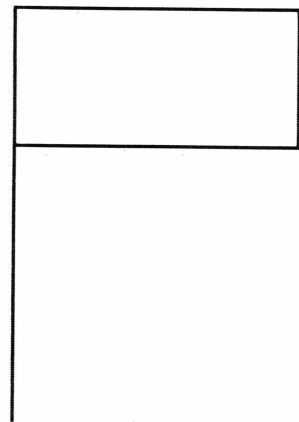


Figure 2.4: Figure drawn by executing FLAG 50.

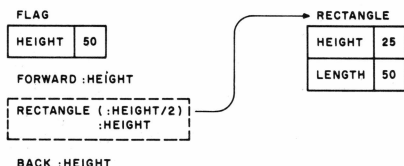


Figure 2.5: Private libraries set up by executing FLAG 50.

⁴ Section 5.7.2 discusses the rules for using parentheses in Logo.

The importance of private input names is that you can use a procedure without concern for the details of precisely *how it is coded*, but rather just concentrating on *what it does*. When you write the FLAG procedure, you can regard RECTANGLE as a “black box” that draws a rectangle, without worrying about what names it uses for its inputs. Indeed, as far as FLAG is concerned, RECTANGLE might have been a primitive included in the Logo system.

The technique of regarding a procedure (even a complex procedure) as a black box whose details you needn’t worry about at the moment is a crucial idea in programming or, indeed, in any kind of design enterprise. Each time you define a new procedure, you can use it as a building block in more complex procedures, and in this way you can build up very complex processes in what Papert [12] refers to as “mind-size bites.”

As a simple illustration, once you have defined FLAG you can use it to easily make a procedure that draws a flag and moves the turtle over a bit:⁵

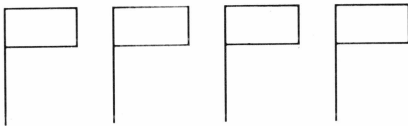


Figure 2.6: Picture drawn by ROW 20 30 4.

```
TO FLAG.AND.MOVE :SIZE :DISTANCE
  PENDOWN
  FLAG :SIZE
  PENUP
  RIGHT 90
  FORWARD :DISTANCE
  LEFT 90
  END
```

and you can use this to draw a row of flags as in figure 2.6:

```
TO ROW :SIZE :SPACING :HOW.MANY
  REPEAT :HOW.MANY
    [FLAG.AND.MOVE :SIZE :SPACING]
  END
```

Type as one line. See page 9. →

⁵ The period used in a name like FLAG.AND.MOVE is interpreted as an ordinary character. Logo does not allow spaces to be part of procedure names, so the period is a useful way to make long names more readable.

2.1.3. An arc procedure

As another example of using procedures with inputs, we'll consider how to use the turtle to draw circles and circular arcs. This project illustrates a number of important ideas, both in programming and in geometry.⁶

The arc procedure will be based on the following fact: making the turtle go FORWARD a small fixed distance, turn a small fixed angle, and repeat this over and over draws a good approximation to a circular arc.⁷ When the turtle has turned through 360 degrees, a complete circle will have been drawn. This leads to the following circle procedure:⁸

```
TO CIRCLE1
REPEAT 360 [FORWARD 1 RIGHT 1]
END
```

You can make this procedure more useful by giving it an input that varies the size of the circle:

```
TO CIRCLE2 :SIZE
REPEAT 360 [FORWARD :SIZE RIGHT 1]
END
```

Note that the turtle still turns one degree at each step, and that varying the size of the FORWARD step varies the size of the circle. Figure 2.7 shows some circles drawn by the CIRCLE2 procedure.

On the other hand, it seems even more useful to have a circle procedure whose size input specifies the radius of the circle that will be drawn. To accomplish this you can reason as follows: the CIRCLE2 procedure draws a circle whose circumference is 360 times SIZE. Since the circumference of a circle is equal to 2π times its radius, we have

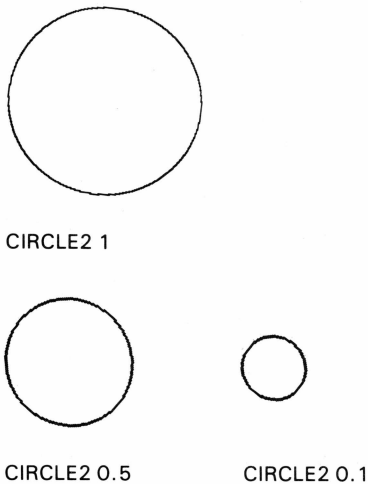


Figure 2.7: Circles drawn by the CIRCLE2 procedure.

⁶ Since circles and arcs are so useful in drawing, Apple Logo comes with pre-defined procedures ARCRIGHT (also written ARCR), ARCLEFT (also written ARCL), CIRCLEAR, CIRCLEL. These are automatically loaded as part of an "aids package" when the Logo system is started. In order not to conflict with the pre-supplied procedures, (which are slightly different from the ones developed below) we'll name our arc procedures NEW.ARCRIGHT and NEW.ARCLEFT. In drawing pictures with arcs, you can use either the procedures given here or the ARCRIGHT and ARCLEFT procedures supplied with Logo.

⁷ This is a fundamental idea in turtle geometry, based on the fact that a circle is a curve of constant curvature. This observation is the key to many turtle-based approaches to mathematics as described in the book by Abelson and diSessa [1].

⁸ The digit 1 included as part of the name CIRCLE1 is interpreted as an ordinary character. It is standard practice to name minor variants of procedures by appending a number to the name.

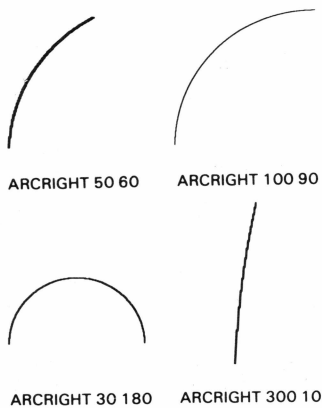
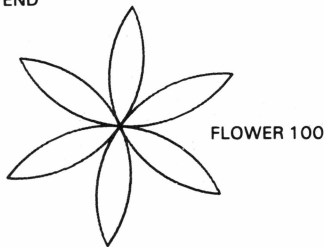


Figure 2.8: Circular arcs drawn by the ARCRIGHT procedure.

```
TO PETAL :SIZE
  ARCRIGHT :SIZE 60
  RIGHT 120
  ARCRIGHT :SIZE 60
  RIGHT 120
END
```

```
TO FLOWER :SIZE
  REPEAT 6 [PETAL :SIZE RIGHT 60]
END
```



```
TO RAY :SIZE
  ARCLEFT :SIZE 90
  ARCRIGHT :SIZE 90
  ARCLEFT :SIZE 90
  ARCRIGHT :SIZE 90
END
```

```
TO SUN :SIZE
  REPEAT 9 [RAY :SIZE RIGHT 160]
END
```

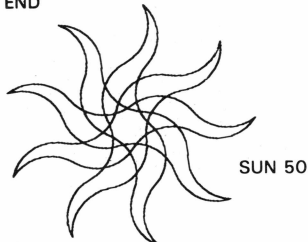


Figure 2.9: Simple procedures that use arcs.

$$360 \times \text{SIZE} = 2\pi \times \text{radius}$$

or

$$\text{SIZE} = \text{radius} \times \pi/180 = \text{radius} \times 0.0174$$

as a good approximation. Thus you can define the desired CIRCLE procedure using CIRCLE2 as a subprocedure:

```
TO CIRCLE :RADIUS
  CIRCLE2 :RADIUS * 0.0174
END
```

The arc procedure can be implemented in the same way, except the turtle should turn only as many 1-degree steps as there are degrees in the arc. The following procedure draws circular arcs turning toward the right:

```
TO NEW.ARCRIGHT :RADIUS :DEGREES
  ARCRIGHT1 :RADIUS * 0.0174 :DEGREES
END
```

```
TO ARCRIGHT1 :SIZE :DEGREES
  REPEAT :DEGREES [FORWARD :SIZE RIGHT 1]
END
```

Figure 2.8 shows some arcs generated by this procedure. A NEW.ARCLEFT procedure can be designed in exactly the same way.⁹

Figure 2.9 shows some examples of drawings that use arc procedures.

Improving the Arc procedure

The NEW.ARCRIGHT procedure given above draws the correct shape, but it is very slow, especially if you use it without hiding the turtle. The problem is that there are so many FORWARD 1, LEFT 1 moves per arc. And these are mostly unnecessary, because, within the accuracy of the display screen, a regular polygon with more than, say, 20 sides is indistinguishable from a circle. For example, you can replace the CIRCLE1 procedure above by the following procedure, which draws a regular 36-sided polygon:

⁹ One drawback of these arc definitions is that the use of REPEAT means that they will only work if the number of DEGREES specified is a non-negative integer. It is not hard to design a better arc procedure that gets around this difficulty.

```

TO CIRCLE1 :SIZE
REPEAT 36 [FORWARD :SIZE * 10 RIGHT 10]
END

```

(Notice that you multiply the FORWARD step by 10 in order to keep the circle the same size as before.) Using this new CIRCLE1, the CIRCLE procedure runs about 10 times as fast and looks almost the same on the display screen.

You can do the same trick with arcs, but there is a complication. You would like to define a new ARCRIGHT1 procedure analogous to the new CIRCLE1, where the turtle turns in units of 10 degrees, something like

```

TO ARCRIGHT1 :SIZE :DEGREES
REPEAT :DEGREES / 10
  [FORWARD :SIZE * 10 RIGHT 10]
END

```

Type as one line. See page 9.—

This works fine as long as the DEGREES input is a multiple of 10. But suppose it is not. For example, if DEGREES were 76, this procedure would draw a 70-degree arc and then stop. You need to add an extra correction step to account for the extra 6 degrees. The correction step should make the turtle go forward a bit and then turn 6 degrees. How far should the turtle go forward? A reasonable guess is to make it go forward $6/10$ times the distance it went forward on the 10-degree turns.¹⁰

In general, given an arbitrary (positive) number for DEGREES you make the turtle do as many

```

FORWARD :SIZE * 10
RIGHT 10

```

steps as the quotient of DEGREES by 10. Then you correct this by having the turtle go forward a distance proportional to the remaining number of degrees, and turn this extra angle. In implementing this procedure, you can use the Logo primitive functions QUOTIENT and REMAINDER that output, respectively, the integer quotient and remainder of their two inputs.¹¹ This yields the procedure

¹⁰ This kind of proportional correction is known as making a linear approximation. In the situation at hand, it is accurate to better than 1 part in 100, so is perfectly adequate for these purposes.

¹¹ For example, QUOTIENT 76 10 is 7 and REMAINDER 76 10 is 6. By contrast, $76 / 10$ is 7.6. For more information on Logo arithmetic operations, see the discussion in section 5.1.

Type as one line. See page 9. →

```
TO ARCRIGHT1 :SIZE :DEGREES
REPEAT (QUOTIENT :DEGREES 10)
  [FORWARD : SIZE * 10 RIGHT 10]
CORRECTARC :SIZE (REMAINDER :DEGREES 10)
END
```

```
TO CORRECTARC :SIZE :AMOUNT
FORWARD :SIZE * :AMOUNT
RIGHT :AMOUNT
END
```

With this new ARCRIGHT1, the NEW.ARCRIGHT procedure, which scales the size so that the input specifies the radius of the arc, is written as on page 30:

```
TO NEW.ARCRIGHT :RADIUS :DEGREES
ARCRIGHT1 :RADIUS * 0.0174 :DEGREES
END
```

You should make one further improvement to the arc procedure. If you use the above procedure to draw a semicircle (DEGREES = 180) starting with the turtle pointing straight up, the semicircle will look slightly tilted. (Try it.) This is because the turtle goes FORWARD (straight up) before doing any turning, so the first small line is vertical. Similarly, the turtle does its final 10-degree turn after drawing the last line, so the last line of the semicircle is not vertical. This asymmetry accounts for the lopsidedness. One way to fix this is to have the turtle turn 5 degrees before doing any drawing and then turn back 5 degrees after the arc is done. The resulting arc is then more symmetrically oriented with respect to the turtle's initial and final headings:

```
TO NEW.ARCRIGHT :RADIUS :DEGREES
RIGHT 5
ARCRIGHT1 :RADIUS * 0.0174 :DEGREES
LEFT 5
END
```

2.2. Repetition and Recursion

We've already seen the use of the Logo REPEAT command (page 9) to repeat some series of steps some fixed number of times. Another way to make something repeat is to define a procedure that includes a call to itself as the final line. For example,

```

TO SQUARE :SIZE
FORWARD :SIZE
RIGHT 90
SQUARE :SIZE
END

```

makes the turtle move in a square pattern over and over again until you stop it by pressing CTRL-G. You can think of the way this procedure works as a kind of joke—the steps of a procedure can include calls to any procedure, so why not call the procedure itself? In this case, the definition of SQUARE is “go forward, turn right, and then do SQUARE again.” And this last step entails going forward, turning right, and then doing SQUARE again, and so on forever.¹²

One disadvantage of this SQUARE as opposed to the one we have been previously using,

```

TO SQUARE :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
END

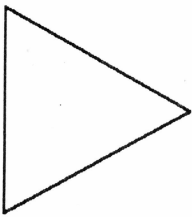
```

is that it goes on indefinitely and so is not a good building block to use in making more complex drawings. On the other hand, this kind of indefinite repetition can be useful in situations in which you do not know (or cannot easily figure out) how many times to repeat some sequence of steps. The following program is an excellent example:

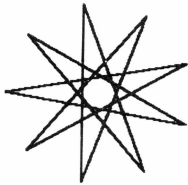
```

TO POLY :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLY :SIDE :ANGLE
END

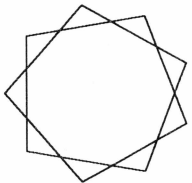
```



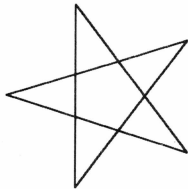
POLY 50 120



POLY 50 160



POLY 60 80



POLY 80 144

Figure 2.10: Shapes drawn by the POLY program.

Figure 2.10 shows some of the many figures drawn by POLY as the angle varies. They are all closed figures, but the number of sides that must be drawn before the figure closes depends in a complicated way upon the ANGLE input to the program.¹³ Using the indefinite repeat you can draw them all with a single simple procedure.

¹² Compare: If a genie appears and offers you three wishes, you should use your third wish to wish for three more wishes.

¹³ This phenomenon forms the basis for a number of mathematical investigations involving symmetry and number theory, described in Abelson and diSessa [1].

Recursion is the programming word for the ability to use the term POLY as part of the definition of POLY, or, in general to write procedures that call themselves.¹⁴

2.2.1. Thinking about recursion

The recursive procedures above have a very simple form—they merely repeat an unchangeable cycle over and over again. Recursion is a much more powerful idea and can be used to obtain much more complicated effects. We shall meet many examples. To take just a small step beyond the purely repetitive kind of recursion, consider

```
TO COUNTDOWN :NUMBER
PRINT :NUMBER
COUNTDOWN :NUMBER - 1
END
```

Let's examine what happens if you give the command

```
COUNTDOWN 10
```

To understand the effect of this command, look back at the definition of the COUNTDOWN procedure. You see that it needs an input and that it uses the name NUMBER for this input. In this case, you have given 10 as the input, so the procedure takes NUMBER to be 10.¹⁵ The first line says

```
PRINT :NUMBER
```

so it prints 10 and goes on to the next line, which is

```
COUNTDOWN :NUMBER - 1
```

or, in this case

```
COUNTDOWN 9
```

This order causes the same effect as if you had typed in the command

```
COUNTDOWN 9
```

¹⁴ Languages like FORTRAN and (most versions of) BASIC do not allow recursion because the implementation of a computer language is simplified if one can assume that there are no recursive functions.

¹⁵ Using the terminology introduced on page 26 we would say that COUNTDOWN sets up a private library in which the name NUMBER is associated with 10.

which would be to print 9 and then give the order

COUNTDOWN 8

and so on. . . In sum, the effect of

COUNTDOWN 10

is to print 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, . . . until you stop the process by typing CTRL-G.

Another example of the same programming technique is the following modification of the POLY program on page 33:

```
TO POLYSPI :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLYSPI (:SIDE + 3) :ANGLE
END
```

Giving the command

POLYSPI 0 90

leads to the sequence of turtle moves

```
FORWARD 0
RIGHT 90
FORWARD 3
RIGHT 90
FORWARD 6
RIGHT 90
FORWARD 9
RIGHT 90
```

.
.

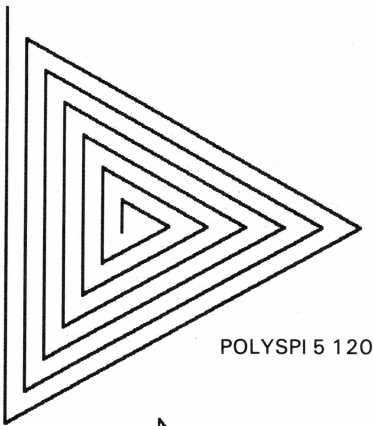
.

.

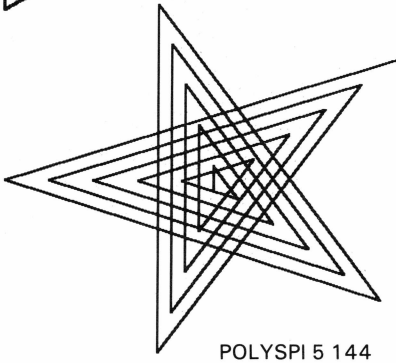
which produces a square-like spiral.¹⁶ By changing the ANGLE input you can draw all sorts of spiral shapes, as shown in figure 2.11. Part of the power of recursion is the fact that such simple programs can lead to such varied, unexpected results.

Suppose you want COUNTDOWN to stop before printing 0. You can do this as follows:

2.2.2. Conditional commands and STOP



POLYSPI 5 120



POLYSPI 5 144

Figure 2.11: Shapes drawn by the POLYSPI program.

¹⁶ Or "squiral," as it was dubbed by a 5th grade Logo programmer who discovered this figure.

```

TO COUNTDOWN :NUMBER
IF :NUMBER = 0 [STOP]
PRINT :NUMBER
COUNTDOWN :NUMBER - 1
END

```

The IF statement is used in Logo to perform tests, in this case to test whether the value of NUMBER is zero. If so, the COUNTDOWN procedure STOPS. That is, rather than continuing with the next line in the procedure, it returns control to wherever the procedure was originally called from. So in response to the command

```
COUNTDOWN 5
```

the computer prints 5, 4, 3, 2, 1 and prompts for a new command.

Keep in mind that the idea of STOP is that when a procedure stops, the next command that gets executed is the one after the command that called the procedure. For example,

```

TO BLASTOFF
COUNTDOWN 10
FORWARD 100
END

```

counts down from 10 to 1 and then moves the turtle.¹⁷

The IF statement is called a *conditional expression*. It has the form

```
IF {some condition is true} {do some action}
```

The action to be performed is specified as a list enclosed in brackets, similar to the list used with REPEAT.

The kinds of conditions that can be tested are generated by Logo operations called *predicates*. Predicates are things whose value is either true or false. COUNTDOWN uses =, which is true if the two things it is comparing are equal. Two other predicates are >, which tests whether the number on its left is greater than the number on its right, and <, which tests for less than. These three predicates deal with numbers.¹⁸ Logo includes other predicates for dealing with other kinds of data.

¹⁷ This stopping behavior is just what normally happens after a procedure executes its final line. If you like, you can imagine that every procedure includes a STOP command at its end.

¹⁸ Actually, = can be used for testing equality of any two pieces of Logo data. A precise description of the behavior of = is given on page 184.

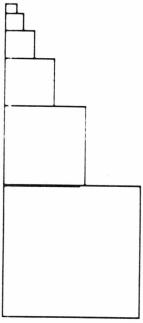


Figure 2.12: Picture drawn by TOWER 50.

2.2.3. Thinking harder about recursion

It is also easy to define your own special-purpose predicates (see section 5.6).

Here is a turtle program based on the COUNTDOWN model. It draws a tower of squares that gets smaller and smaller and stops when the squares get very tiny, as shown in figure 2.12.

```
TO TOWER :SIZE
  IF :SIZE < 3 [STOP]
  SQUARE :SIZE
  FORWARD :SIZE
  TOWER :SIZE * .6
END
```

The recursion examples we have seen so far, in which the recursive call is the final step in the procedure, can be readily viewed as a kind of generalized repetition.¹⁹ Other uses of recursion can be much more powerful, but, unfortunately, much harder to understand. Let's compare the COUNTDOWN procedure from page 36

```
TO COUNTDOWN :NUMBER
  IF :NUMBER = 0 [STOP]
  PRINT :NUMBER
  COUNTDOWN :NUMBER - 1
END
```

with the following similar-looking procedure:

```
TO MYSTERY :NUMBER
  IF :NUMBER = 0 [STOP]
  MYSTERY :NUMBER - 1
  PRINT :NUMBER
END
```

As we saw,

```
COUNTDOWN 3
```

prints 3, 2, 1. In contrast

```
MYSTERY 3
```

¹⁹ The special case of recursion in which the recursive call is the final step is sometimes called *tail recursion*. Logo includes techniques for implementing tail recursion efficiently, so that a tail recursive procedure can effectively run "forever" without running out of storage.

prints 1, 2, 3. Most people find this very hard to understand.

Let's trace through the process carefully. You first call MYSTERY with the input 3, so, as explained on page 26, MYSTERY sets up a private library in which NUMBER is associated with 3. It checks whether the value of NUMBER is 0, which it is not, so MYSTERY proceeds to the next line which produces the command

MYSTERY 2

Now let's stop and think. Eventually this second MYSTERY call will stop, and the original

MYSTERY 3

procedure will have to continue with the next command after the call. But this means that there will have to be, in some sense, *two* MYSTERY procedures existing at once—the one called by the command

MYSTERY 2

and the original one called by the command

MYSTERY 3

which is waiting for the other MYSTERY to stop, so it can continue. Moreover, each MYSTERY has its *own* value for NUMBER—NUMBER is 2 for one and 3 for the other. Each MYSTERY must maintain a *separate* private library.²⁰ The situation is shown in figure 2.13.

Let's go on. The first thing that the

MYSTERY 2

procedure does is check whether the value for NUMBER is equal to 0. Since this is not the case, MYSTERY gives the command

MYSTERY 1

and so now there are *three* MYSTERY procedures! And this

MYSTERY 1

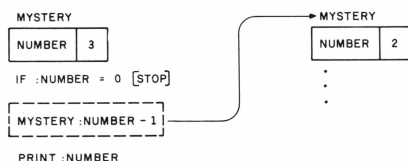


Figure 2.13: Beginning execution of MYSTERY 3.

²⁰ In other words, the private library is associated, not with a procedure, but with a given call to a procedure (or what is technically called an *activation* of a procedure).

does a test and calls up yet another

MYSTERY 0

which makes four MYSTERY calls all existing at once as shown in figure 2.14. Note that so far nothing has been printed. All that has happened is that MYSTERY procedures have called up more MYSTERY procedures.

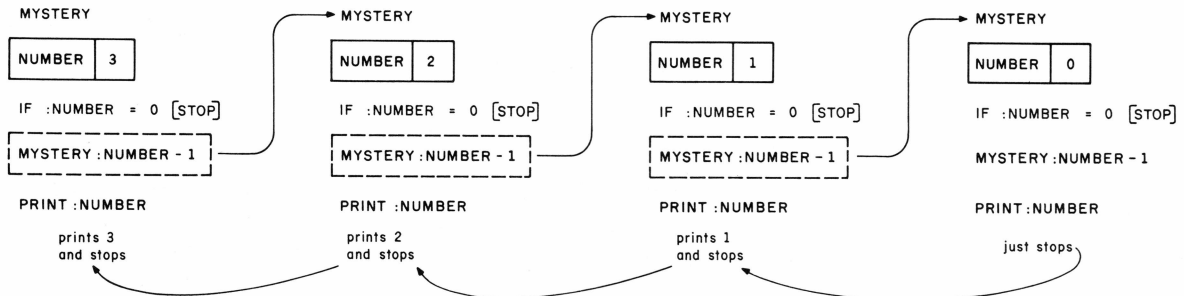


Figure 2.14: Complete execution of MYSTERY 3.

Now

MYSTERY 0

performs its test and finds that the value of NUMBER is indeed 0. So it STOPS and the process continues with the procedure that called it, namely,

MYSTERY 1

This MYSTERY now proceeds with the next line after the call, which says to print the value of NUMBER. Since NUMBER is 1 (in *this* MYSTERY's private library) it prints 1. Now it is done and so returns to the procedure that called it, namely

MYSTERY 2

This MYSTERY now continues with the line after the call, which says to print NUMBER. Since NUMBER is 2 (in *this* private library) it prints 2 and returns to its caller, namely,

MYSTERY 3

which prints 3 and returns to its caller, which is the main Logo command level.

Whew! Try going through this example again step by step, referring to figure 2.14. In essence, this complex process is doing nothing more than unwinding the following rule:

- When a procedure is called, the calling procedure waits until the second procedure stops, and then continues with the next instruction after the call.

Recursion, however, forces us to appreciate all the ramifications of this simple sounding rule. In particular,

- There may be several instances (or “activations”) of the “same” procedure all coexisting at once.
- Each procedure activation has a *separate* private library, so the “same” name may be associated with different values in different procedure activations.
- The order in which things happen can be very confusing.²¹

2.2.4. Drawing trees

As another example of complex use of recursion, let’s look at a program that draws a *binary tree* as in figure 2.15.

Think about how you would describe this figure. One way to do it would be to say something like “the tree is a vee-shape with a smaller tree at each tip. And each smaller tree is a vee-shape with a still smaller tree at each of its tips, and so on.” This is a *recursive description* of the tree. You can translate this description into a *recursive procedure* that draws the figure. You start with the following commands that make the turtle draw a vee-shape of a certain length and return to its initial position and heading:

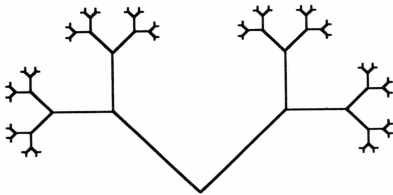


Figure 2.15: A binary tree.

```
LEFT 45
FORWARD :LENGTH
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
BACK :LENGTH
LEFT 45
```

This is the basic vee-shape of the tree. Now, according to the recursive description, the entire tree consists of this vee with smaller vees (say, half as big) drawn at each tip. So the TREE procedure should be something like

²¹ More specifically, things happen in the reverse order from the way one might expect. This is a consequence of the fact that the last procedure called is the first one to stop.

```

TO TREE :LENGTH
LEFT 45
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
LEFT 45
END

```

But this doesn't quite work. Consider—if you call TREE with an input of 20, this will make the turtle go LEFT 45, FORWARD 20 and call

```
TREE 10
```

which will make the turtle go LEFT 45, FORWARD 10 and call

```
TREE 5
```

and so on forever.²² This is something like the forever-running COUNTDOWN procedure on page 34, or even more like the chain of MYSTERY procedures on page 38, in that no procedure finishes until the last one to be called has stopped. What you need is a *stop rule* to keep the process from going on forever. You can make the process stop by having the procedure just stop without drawing anything if LENGTH is very small:

```

TO TREE :LENGTH
IF :LENGTH < 2 [STOP]
LEFT 45
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
LEFT 45
END

```

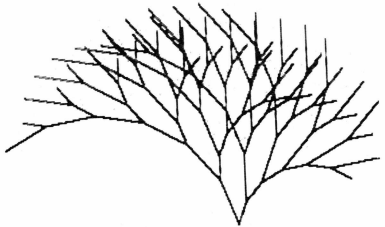
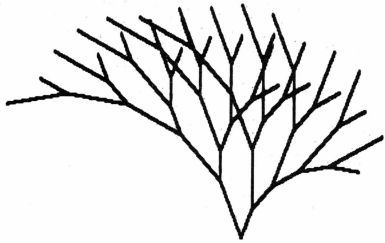
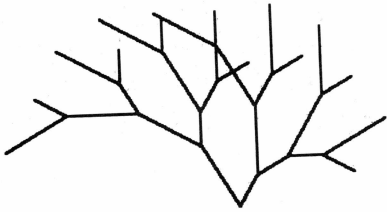
²² That is, until Logo runs out of storage.

You can modify the TREE procedure to produce a procedure TREE1, in which the subtree branches have the same length as the original branches, rather than half the length. If you do this, however, then the branches of successive subtrees will not get smaller and smaller, which means that you cannot use the same stop rule as in TREE. A different strategy for providing a stop rule is to include for TREE1 an extra input, DEPTH, which determines the “depth” to which the tree is drawn. Each tree of a given depth spawns two subtrees of depth one less. When the TREE procedure is called with DEPTH equal to 0, it just stops without drawing:

```
TO TREE1 :LENGTH :DEPTH
IF: DEPTH = 0 [STOP]
LEFT 45
FORWARD :LENGTH
TREE1 :LENGTH :DEPTH - 1
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE1 :LENGTH :DEPTH - 1
BACK :LENGTH
LEFT 45
END
```

Thinking in terms of recursive descriptions can take a lot of getting used to, and the programs can be subtle. One especially subtle point about the TREE program is the final BACK and LEFT moves, which are needed to restore the turtle to its initial heading so that the different calls to TREE will fit together correctly. On the other hand, many seemingly complex designs have simple recursive descriptions and can be drawn by remarkably brief programs. The design of recursive turtle programs for drawing complex patterns is discussed extensively in Abelson and diSessa [1].

To illustrate the flavor of recursive designs, here is a modification to TREE1, in which the left branch of each vee is twice as long as the right branch. We'll also allow the angle of the vee to be varied as an input. Figure 2.16 shows some of the patterns that result.



```

TO NEW.TREE :LENGTH :ANGLE :DEPTH
IF :DEPTH = 0 [STOP]
LEFT :ANGLE
FORWARD 2 * :LENGTH
NEW.TREE :LENGTH :ANGLE :DEPTH - 1
BACK 2 * :LENGTH
RIGHT 2 * :ANGLE
FORWARD :LENGTH
NEW.TREE :LENGTH :ANGLE :DEPTH - 1
BACK :LENGTH
LEFT :ANGLE
END

```

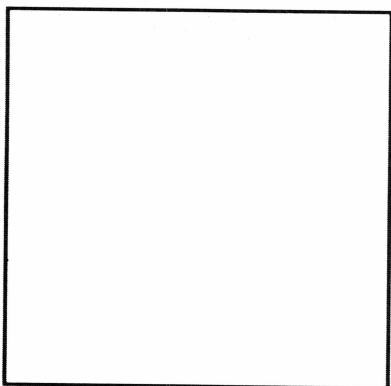
Figure 2.16: Some figures drawn by the NEW.TREE procedure.

CHAPTER 3

Projects in Turtle Geometry

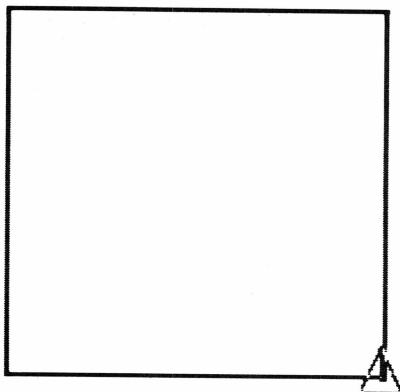
Here are some projects that use Turtle Geometry. Refer to other portions of this text for help in defining or editing programs. Feel free to change programs that are offered and to design new programs.

Here is a square procedure.



```
TO SQUARE
REPEAT 4 [FORWARD 60 RIGHT 90]
END
```

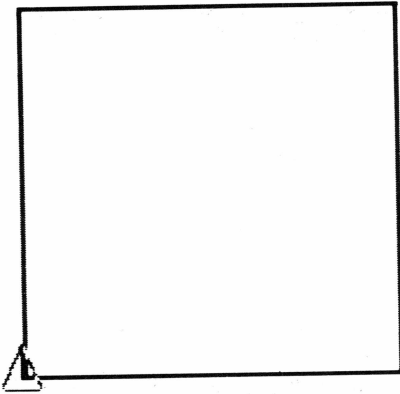
Here are two square procedures designed to allow variable size. The dark arrows show the turtle's initial position.



```
TO LSQUARE :SIZE
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
END
```

or

```
TO LSQUARE :SIZE
REPEAT 4 [FORWARD :SIZE LEFT 90]
END
```

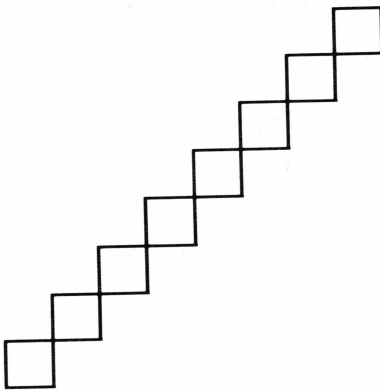


```
TO RSQUARE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
END
```

or

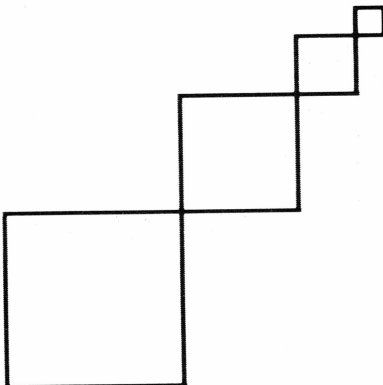
```
TO RSQUARE :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
END
```

**Some procedures using
RSQUARE and recursion.**

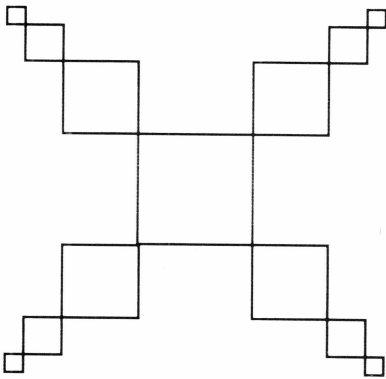


```
TO MOVE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
LEFT 90
END
```

```
TO STAIRS :SIZE
RSQUARE :SIZE
MOVE :SIZE
STAIRS :SIZE
END
```

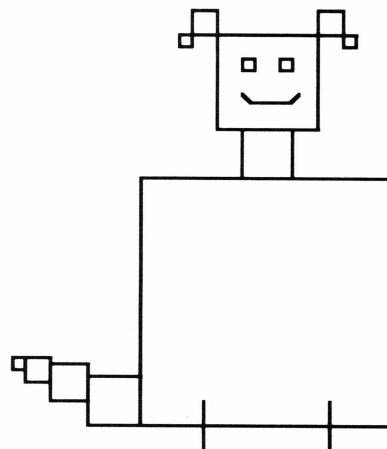
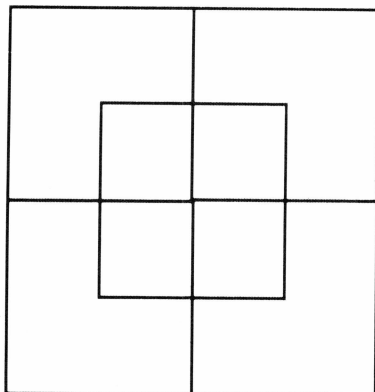
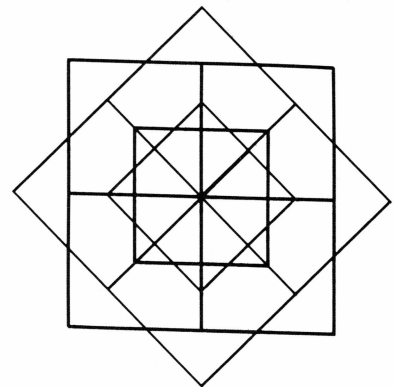
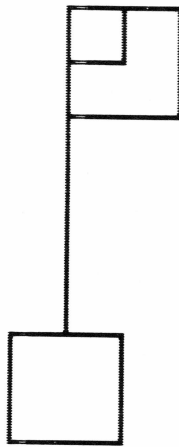
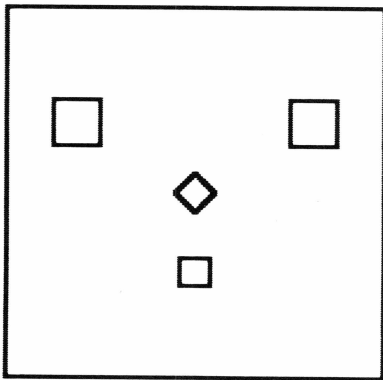


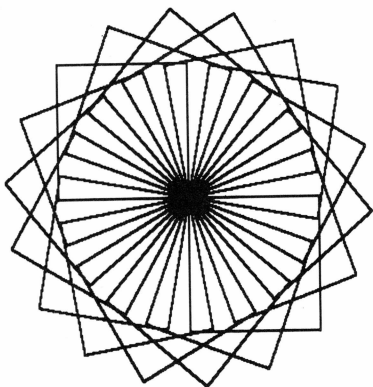
```
TO BOXES
RSQUARE 30
MOVE 30
RSQUARE 20
MOVE 20
RSQUARE 10
MOVE 10
RSQUARE 5
RIGHT 180
PENUP
MOVE 60
RIGHT 180
PENDOWN
END
```



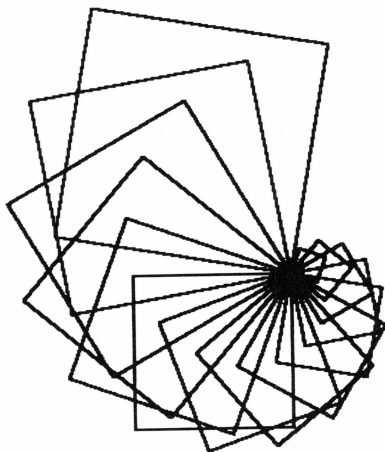
TO MANYBOXES
BOXES
FORWARD 30
RIGHT 90
MANYBOXES
END

Some ideas for using square
procedures.



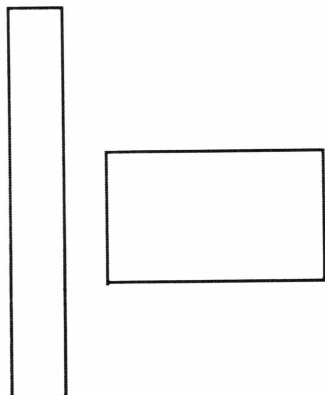


```
TO SPINSQUARES :SIZE
  RSQUARE :SIZE
  RIGHT 20
  SPINSQUARES :SIZE
END
```

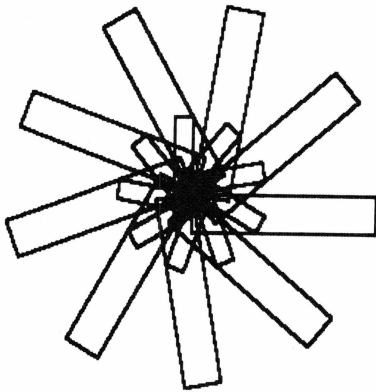


```
TO GROWSQUARES :SIZE
  RSQUARE :SIZE
  RIGHT 20
  GROWSQUARES :SIZE + 5
END
```

A rectangle procedure designed to allow variable size and some examples that use it.



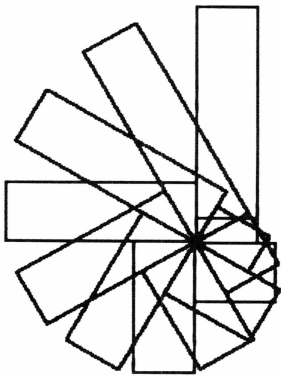
```
TO RECTANGLE :LENGTH :WIDTH
  FORWARD :LENGTH
  RIGHT 90
  FORWARD :WIDTH
  RIGHT 90
  FORWARD :LENGTH
  RIGHT 90
  FORWARD :WIDTH
  RIGHT 90
END
```



```

TO FLOWER
  RECTANGLE 100 20
  RIGHT 20
  RECTANGLE 10 40
  RIGHT 20
  FLOWER
END

```

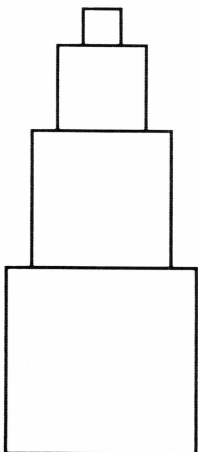


```

TO SPINRECS :SIZE
  IF :SIZE < 10 [STOP]
  RECTANGLE :SIZE 20
  LEFT 30
  SPINRECS :SIZE - 5
END

```

Examples using RSQUARE and RECTANGLE.



```

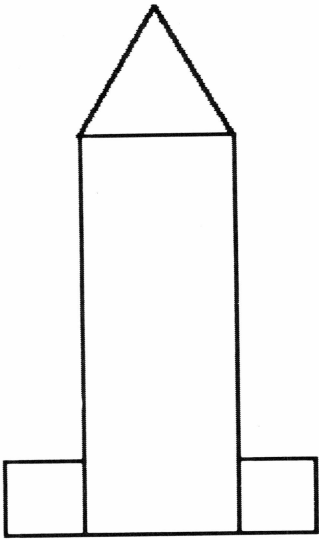
TO HOP :SIZE
  FORWARD :SIZE
  RIGHT 90
  FORWARD 3
  LEFT 90
END

```

```

TO TELESCOPE :SIZE
  IF :SIZE < 6 [STOP]
  RSQUARE :SIZE
  HOP :SIZE
  TELESCOPE :SIZE - 6
END

```

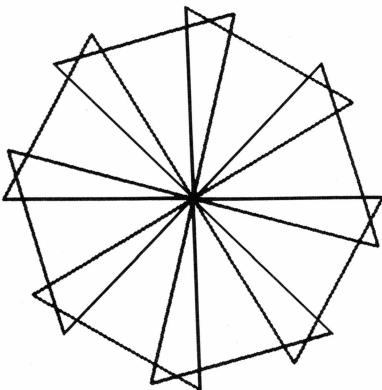
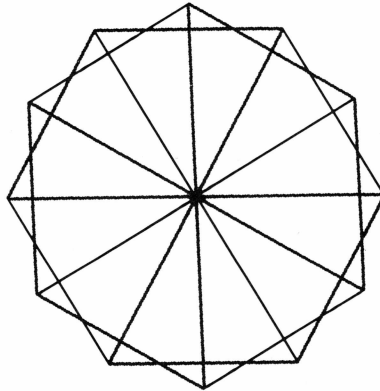
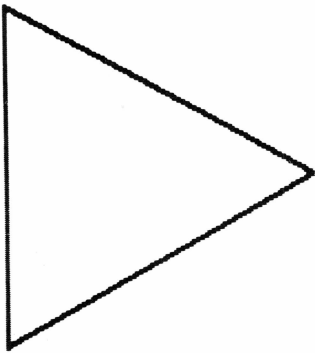



```
TO ROCKTOP  
LEFT 30  
FORWARD 30  
LEFT 120  
FORWARD 30  
END
```

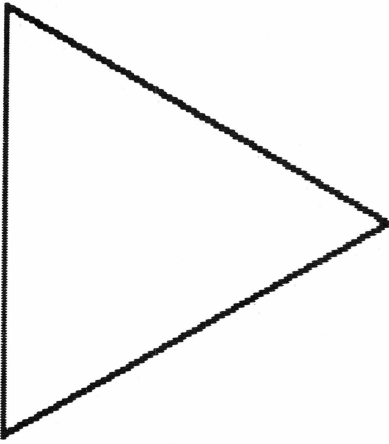
```
TO ROCKET  
RECTANGLE 80 30  
LEFT 90  
RECTANGLE 15 15  
BACK 30  
RIGHT 90  
RECTANGLE 15 15  
FORWARD 80  
ROCKTOP  
END
```

**Here are some examples that use
a triangle procedure.**

```
TO TRI  
REPEAT 3 [FORWARD 70 RIGHT 120]  
END
```

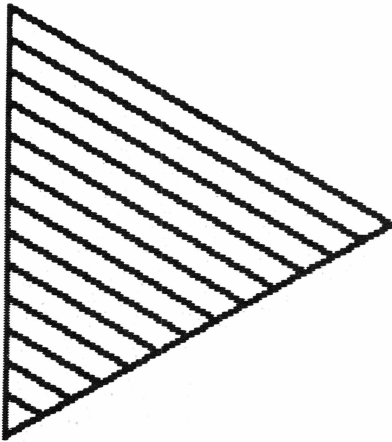


A triangle procedure designed to allow variable size and an example that uses it.



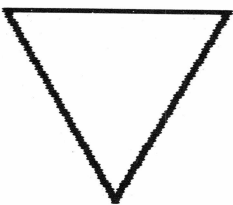
```
TO TRIANGLE :SIZE
FORWARD :SIZE
RIGHT 120
FORWARD :SIZE
RIGHT 120
FORWARD :SIZE
RIGHT 120
END
```

This procedure is different in design but has a similar result.



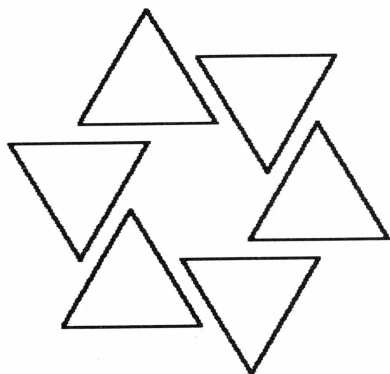
```
TO FLUFF :SIZE
IF :SIZE < 10 [STOP]
TRIANGLE :SIZE
FLUFF :SIZE - 10
```

Some more triangle examples.

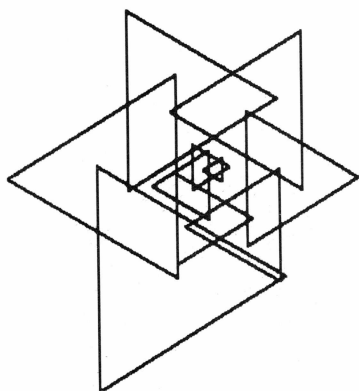


```
TO NEWTRIANGLE :SIZE
LEFT 30
TRIANGLE :SIZE
RIGHT 30
END
```

```
TO CREEP :SIZE
PENUP
FORWARD :SIZE
PENDOWN
END
```



```
TO LOOPS :SIZE
NEWTRIANGLE :SIZE
CREEP :SIZE
RIGHT 60
LOOPS :SIZE
END
```



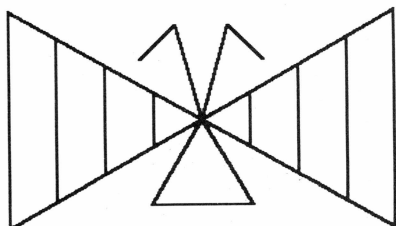
```
TO NEWLOOP :SIZE
IF :SIZE < 20 [STOP]
NEWTRIANGLE :SIZE
CREEP :SIZE/2
RIGHT 60
NEWLOOP :SIZE - 5
END
```

And one more.

```
TO LEFTANT
LEFT 15
FORWARD 30
LEFT 120
FORWARD 15
BACK 15
RIGHT 120
BACK 30
RIGHT 15
END
```

```
TO RIGHTANT
RIGHT 15
FORWARD 30
RIGHT 120
FORWARD 15
BACK 15
LEFT 120
BACK 30
LEFT 15
END
```

```
TO ANTS
RIGHTANT
LEFTANT
END
```



```
TO BUTTERFLY
RIGHT 60
WING
RIGHT 180
WING
RIGHT 120
ANTS
RIGHT 150
TRIANGLE 30
END
```

```

TO WING
TRIANGLE 80
TRIANGLE 60
TRIANGLE 40
TIRANGLE 20
END

```

RCP and LCP are abbreviations for “right circle piece” and “left circle piece”. RARC and LARC stand for right arc and left arc. A circle can be made from pieces of either left or right arcs, leaving the turtle at the left-most or right-most point of the circle.

```

TO RCP :R
RIGHT 5
FORWARD :R * (3.14159/18)
RIGHT 5
END

```

```

TO LCP :R
LEFT 5
FORWARD :R * (3.14159/18)
LEFT 5
END

```

```

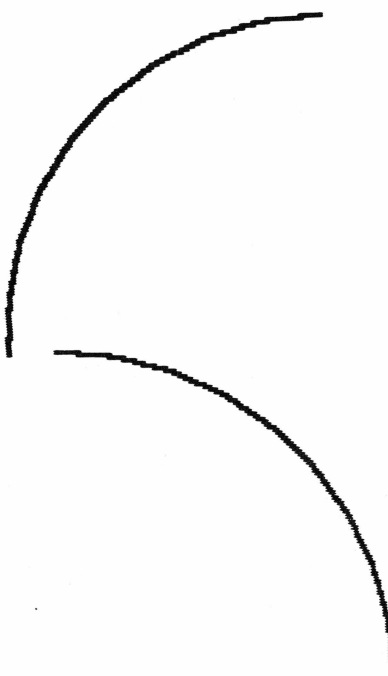
TO RARC :R
REPEAT 9 [RCP :R]
END

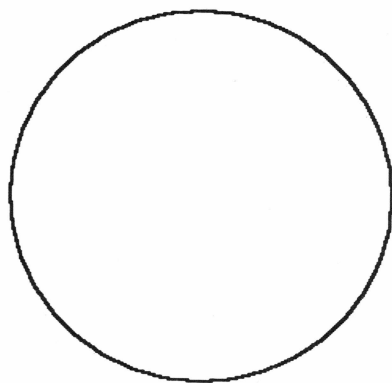
```

```

TO LARC :R
REPEAT 9 [LCP :R]
END

```

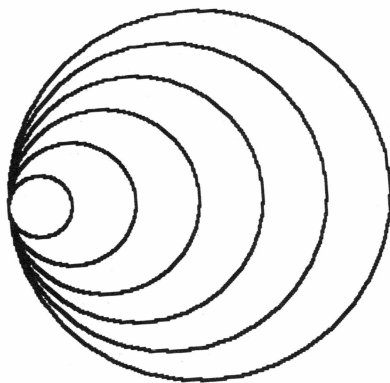




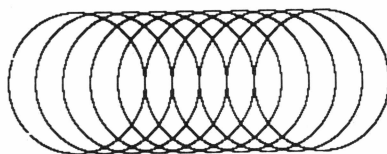
```
TO RCIRCLE :R
REPEAT 36 [RCP :R]
END
```

```
TO LCIRCLE :R
REPEAT 36 [LCP :R]
END
```

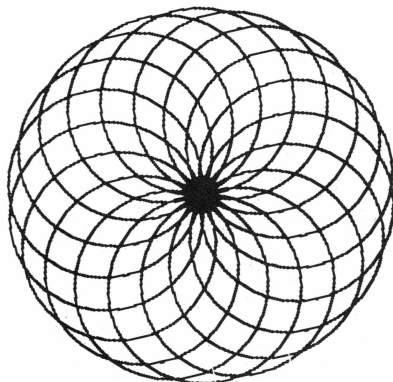
Examples using circle procedures.



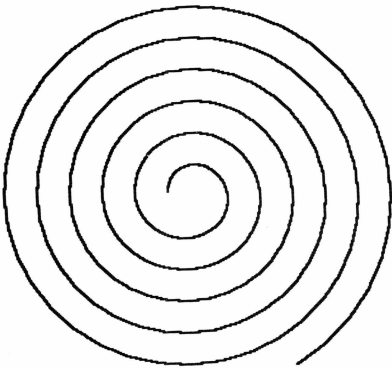
```
TO SHRINKRCIRCLE :SIZE
IF :SIZE < 4 [STOP]
RCIRCLE :SIZE
SHRINKRCIRCLE :SIZE - 2
END
```



```
TO RSLINKY :SIZE
RCIRCLE :SIZE
PU RT 90 FD 10 LT 90 PD
RSLINKY :SIZE
END
```

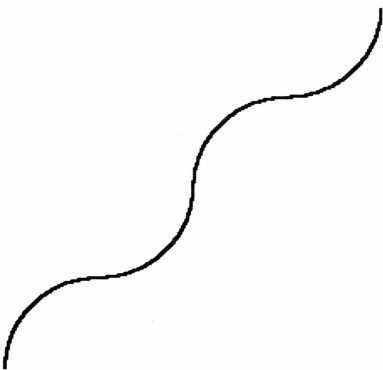


```
TO SPINSLINK :SIZE
RCIRCLE :SIZE
RIGHT 20
SPINSLINK :SIZE
END
```

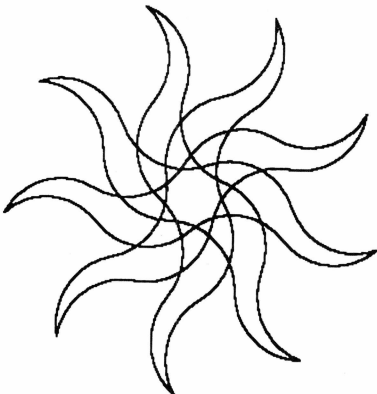


```
TO GROWCIRCLE :SIZE
REPEAT 12 [RCP :SIZE]
GROWCIRCLE :SIZE + 1
END
```

Examples using RARC and LARC.

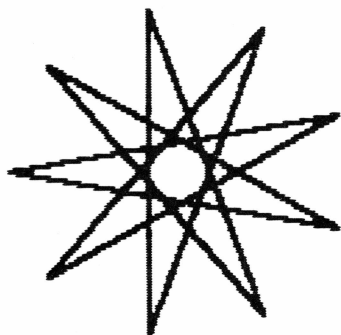


```
TO RAY :SIZE
RARC :SIZE
LARC :SIZE
RARC :SIZE
LARC :SIZE
END
```



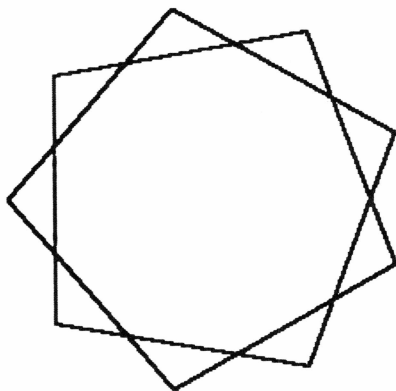
```
TO SUN :SIZE
RAY :SIZE
RIGHT 160
SUN :SIZE
END
```

POLY procedures have variable size and angle. Here are some examples.

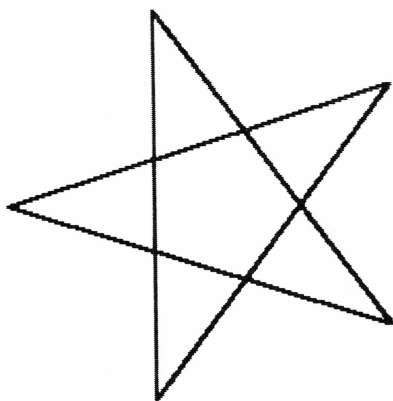


```
TO POLY :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLY :SIDE :ANGLE
END
```

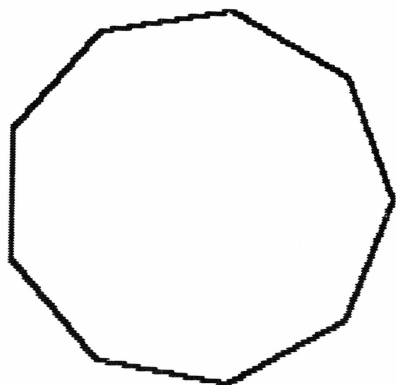
SIDE = 50 ANGLE = 160



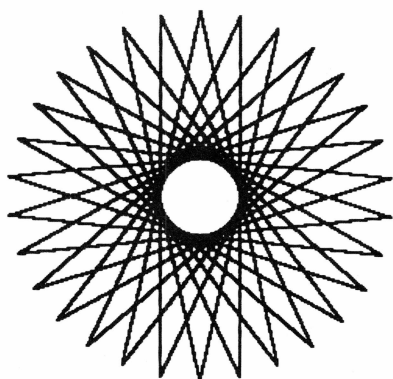
SIDE = 60 ANGLE = 80



SIDE = 80 ANGLE = 144

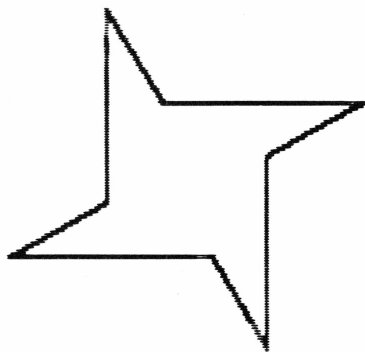


SIDE = 20 ANGLE = 40



SIDE = 100 ANGLE = 156

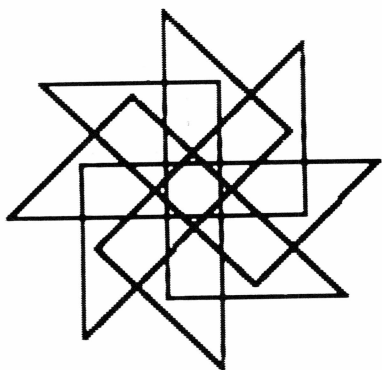
POLYSTEP is a piece of a **POLY** procedure. Here are some examples using it.



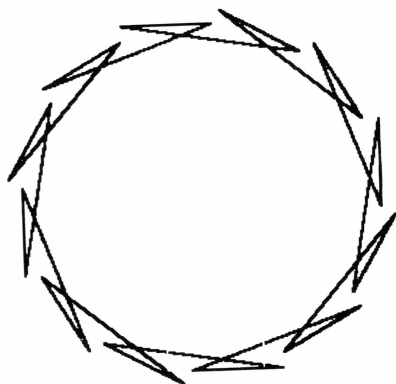
```
TO POLYSTEP :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
END
```

```
TO TWOPOLY :SIDE1 :ANGLE1 :SIDE2 :ANGLE2
POLYSTEP :SIDE1 :ANGLE1
POLYSTEP :SIDE2 :ANGLE2
TWOPOLY :SIDE1 :ANGLE1 :SIDE2 :ANGLE2
END
```

SIDE1 = 30 ANGLE1 = 60 SIDE2 = 60 ANGLE2 = 210



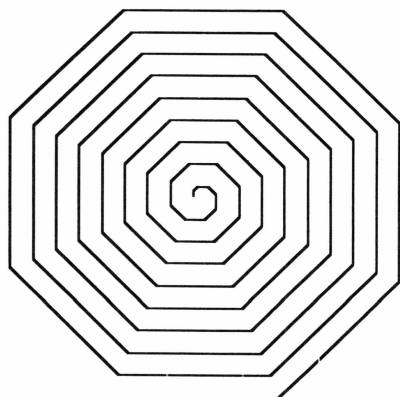
$SIDE1 = 30$ $ANGLE1 = 90$ $SIDE2 = 50$ $ANGLE2 = 135$



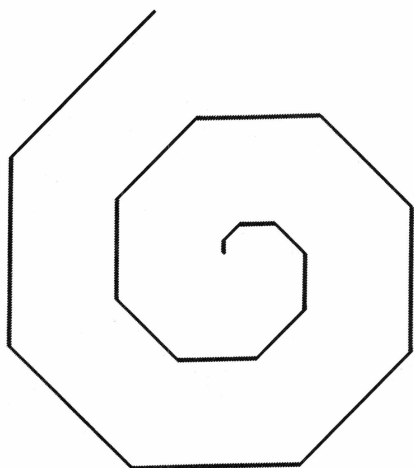
$SIDE1 = 25$ $ANGLE1 = 190$ $SIDE2 = 50$ $ANGLE2 = 200$

**More programs using
POLYSTEP.**

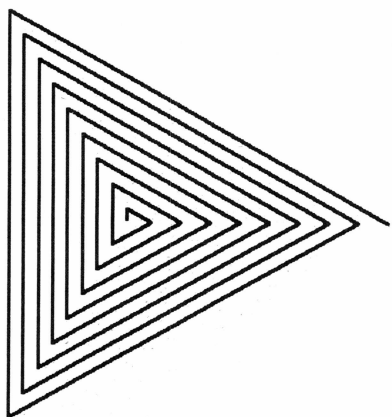
```
TO POLYSPIRAL :SIDE :ANGLE :INC
POLYSTEP :SIDE :ANGLE
POLYSPIRAL (:SIDE + :INC) :ANGLE :INC
END
```



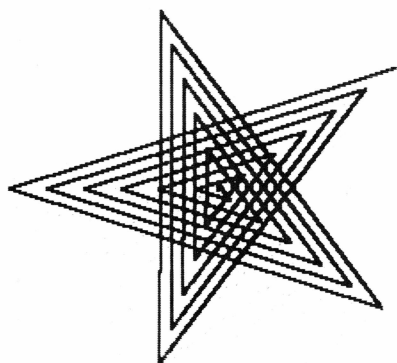
$SIDE = 1$ $ANGLE = 45$ $INCREMENT = 1$



SIDE = 1 ANGLE = 45 INCREMENT = 3

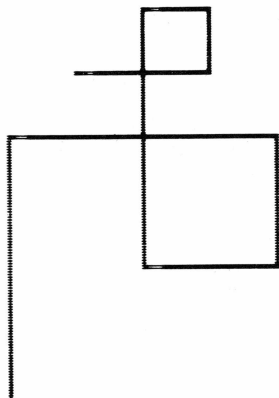


SIDE = 5 ANGLE = 120 INCREMENT = 3

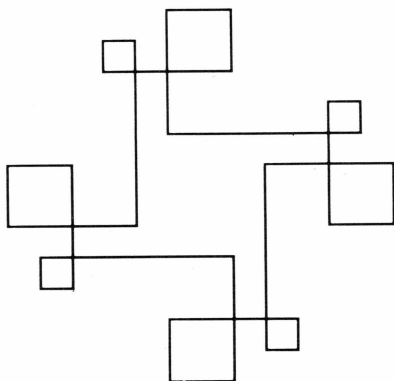


SIDE = 5 ANGLE = 144 INCREMENT = 3

Here's an example that begins by defining a shape and uses it to make a more interesting shape.



```
TO SHAPE
FORWARD 40
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 20
END
```

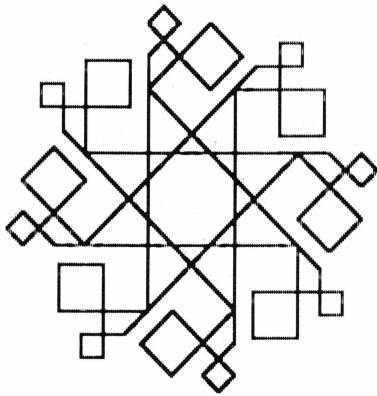


```
TO SHAPE4
SHAPE
SHAPE
SHAPE
SHAPE
END
```

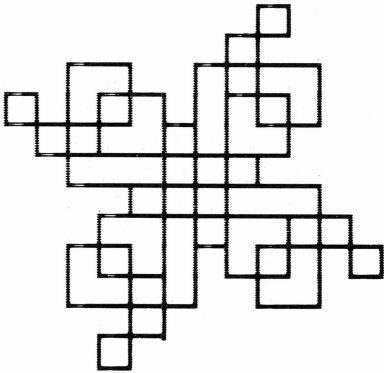
or

```
TO SHAPE4
REPEAT 4 [SHAPE]
END
```

And two more shapes.

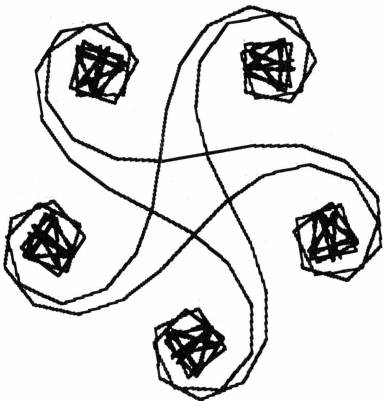


```
TO CRYSTAL
  SHAPE
  LEFT 45
  FORWARD 70
  CRYSTAL
END
```



```
TO JENGU
  SHAPE
  SHAPE
  LEFT 90
  JENGU
END
```

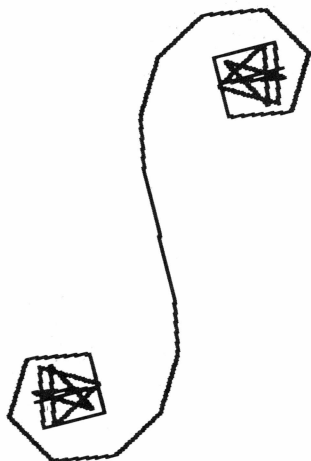
Here are some programs using
INSPI. Try various inputs.



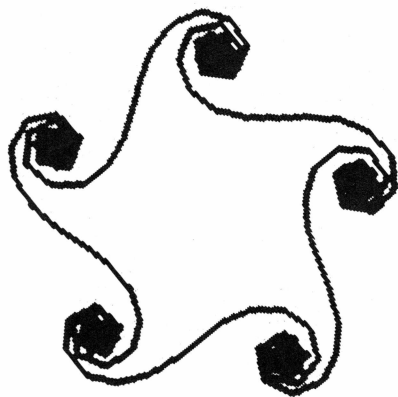
```
TO INSPI :SIDE :ANGLE
  POLYSTEP :SIDE :ANGLE
  INSPI :SIDE (:ANGLE + 10)
END
```

SIDE = 10 ANGLE = 1

More INSPI shapes.



SIDE = 10 ANGLE = 10



SIDE = 7 ANGLE = 3

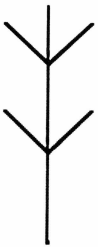


SIDE = 7 ANGLE = 5

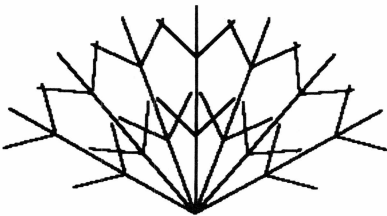
Here is a sequence of procedures that starts with a leaf (VEE) and ends with a forest (TREES).



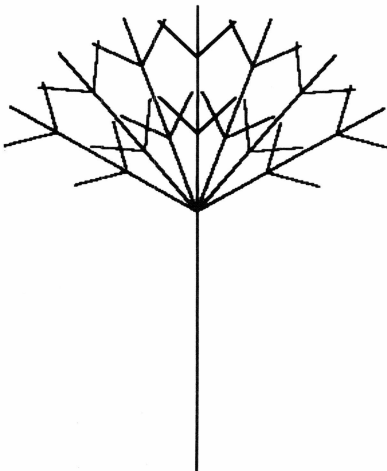
TO VEE
LEFT 45
FORWARD 10
BACK 10
RIGHT 90
FORWARD 10
BACK 10
LEFT 45
END



TO BRANCH
FORWARD 15
VEE
FORWARD 15
VEE
FORWARD 10
BACK 40
END



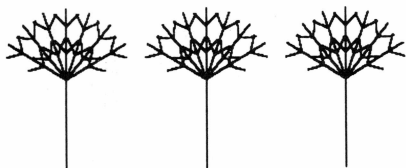
TO BUSH
LEFT 60
REPEAT 6 [BRANCH RIGHT 20]
BRANCH
LEFT 60
END



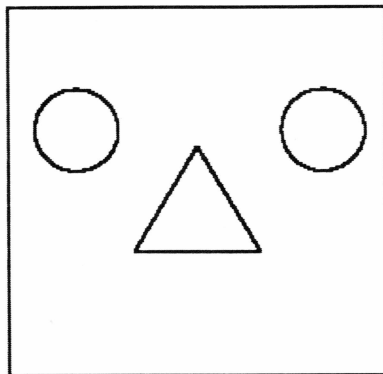
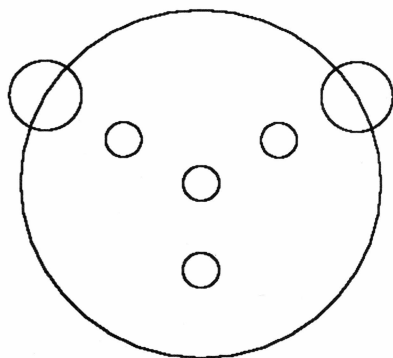
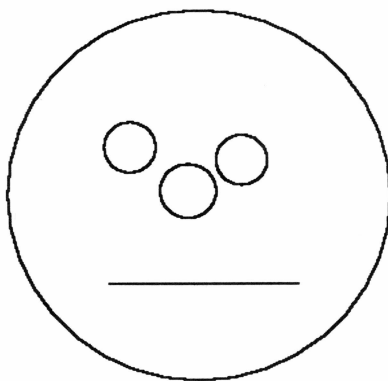
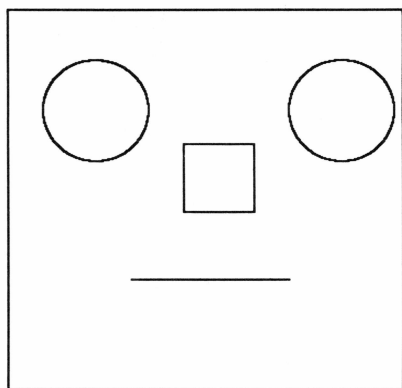
TO GREENTREE
FORWARD 50
BUSH
BACK 50
END

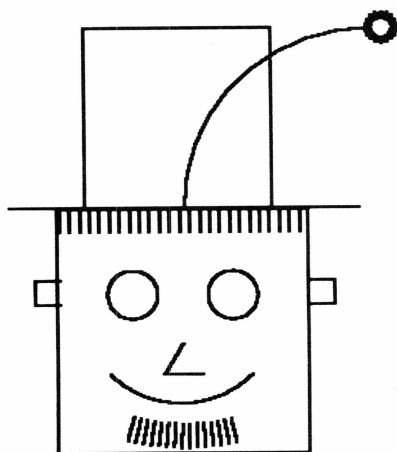
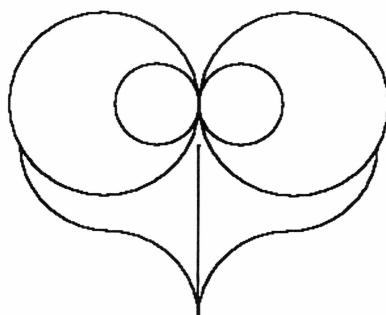
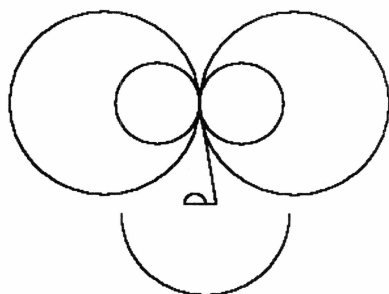
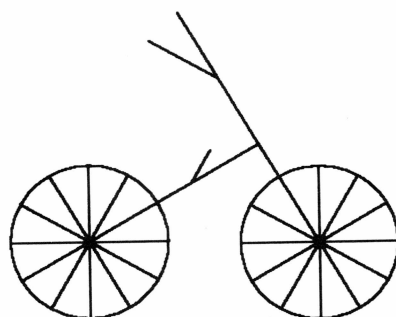
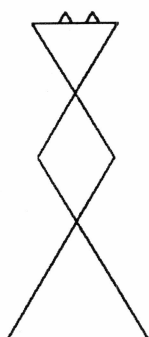
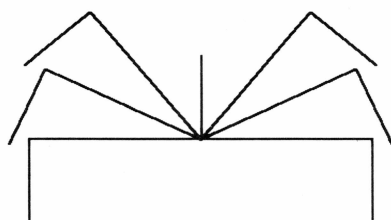
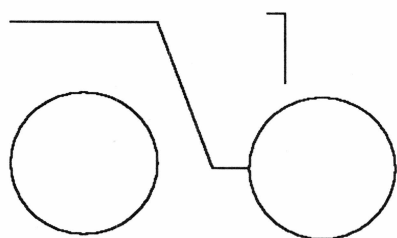
```
TO MOVE
PENUP
RIGHT 90
FORWARD 80
LEFT 90
PENDOWN
END
```

```
TO TREES
REPEAT 3 [GREENTREE MOVE]
END
```



Here are some project ideas using the procedures you have already seen. Feel free to make up your own projects.







CHAPTER 4

Workspace, Filing, and Debugging

When you use the Logo system, you can think of the computer as having two memories. The first memory, called *workspace*, is where Logo keeps track of the procedures and variables you have defined. Each time you define a procedure or assign a value to a name, that information becomes part of the workspace. When you turn off the computer, the information in the workspace is destroyed. The second, and more permanent memory consists of files that you save on disk. Each file contains a complete workspace. The normal way of using the Logo file system is to work for a while on a project and then, when you are finished for the day, to save your workspace in a disk file. The next time you use Logo, you can read in your saved workspace from the disk and continue where you left off. You can also maintain a number of different files containing the workspaces for different projects you are working on.

4.1. Managing Workspace

The Logo system includes commands for examining and deleting various parts of the workspace. These are useful for keeping track of what procedures are currently defined and for getting rid of unwanted definitions.

4.1.1. PO

The basic command for examining workspace is PO. If you type PO followed by the name of a procedure, Logo prints the definition of the procedure. PO can also take a list of procedure names as input, in which case it prints the definitions of all procedures in the list. For example

PO "SQUARE	Makes Logo print the definition of SQUARE.
PO [SQUARE TRIANGLE]	Makes Logo print the definitions of SQUARE and TRIANGLE.

PO also has a few variants:

POTS Prints out the titles of all procedures in workspace.

PONS Prints the names and associated values in the global library.

POPS Prints the definitions of all procedures in workspace.

POALL Prints both names and procedures.

4.1.2. ERASE

The ERASE command (abbreviated ER) is used to get rid of parts of the workspace. ERASE followed by a procedure name removes the definition of the procedure. ERASE can also take as input list of procedures to be erased.

The following variants of ERASE are similar to those for PO:

ERNS Erases the names and associated values from the global library.

ERPS Erases the definitions of all procedures from workspace.

ERALL Erases all names and procedures.

4.1.3. Packages

As an alternative to dealing with your workspace “all in one chunk,” Apple Logo allows you to divide workspace into groups called *packages*. For instance, the command

PACKAGE “GRAPHICS “SQUARE

tells Logo that SQUARE is to be included in a package called GRAPHICS. SQUARE here is presumably a procedure you have defined, but it could also be a name in the global library.

You can use PO and ERASE to work with one package at a time. For example,

POALL “GRAPHICS

will print out all the procedure and names that are part of the package called GRAPHICS, while

ERALL “GRAPHICS

will erase from workspace everything that was declared to be part of the GRAPHICS package.

Burying a package

One particularly convenient use of packages is in conjunction with the BURY command. If you BURY a package, then the information in that package will not normally be affected by commands that “look at the whole workspace,” such as ERALL and POALL. For example, suppose you wish to provide a set of

Logo procedures to be used as programming aids by beginning users, but you would like these procedures to be “invisible” as far as the beginning users are concerned. To do this, define your procedures and place them in a package called, say, PROGRAM.AIDS. Then execute the command

BURY “PROGRAM.AIDS

The procedures included in this buried package can be called as usual, but they will not be printed out if the user types POALL, and they will not be erased if the user types ERALL.¹ At the same time, you can print out or erase these procedures by calling POALL or ERALL with the package name PROGRAM.AIDS as input, or you can PO or ERASE any individual procedure in the package by referring to its name explicitly. Once a package has been buried, you can “unbury” it by using the UNBURY command. See the technical manual for more details on workspace management with packages.

4.1.4. Other uses of EDIT

We introduced simple uses of the Logo EDIT command in section 1.3.2. The general behavior of EDIT is to place the text of a procedure into a storage area called the *edit buffer* where it can be manipulated using the text editing operations provided by the Logo system. When you exit the editor using CTRL-C, the newly edited text is processed and installed in the workspace. It is occasionally useful to load the edit buffer with parts of workspace other than a single procedure definition. To allow this, the EDIT command can take a list of procedure names as input. There is also

EDNS Places Logo in edit mode. The buffer will contain MAKE statements for all of the names in the global library.

For example, if you have a variable A whose associated value is 5 and a variable B whose associated value is 3, the text appearing in the edit buffer will be

```
MAKE "A 5
MAKE "B 3
```

4.2. The Logo File System

The Logo file system allows you to save procedure definitions on floppy disk. A user may have many files on a single disk, and the files are distinguished by the fact that they are named. The names of the files are listed in the disk *catalog*.

¹ In fact, when Apple Logo is started, it sets up just such a buried package called AIDS, which contains procedures for drawing circles and arcs.

4.2.1. Disk files

When you use Logo, you should normally have a Logo file disk mounted in the disk drive. The technical manual describes how to create Logo file disks.

Use the SAVE command to save your procedure definitions on disk. For example, typing

```
SAVE "RECTANGLES
```

saves all the variables and procedure definitions currently in workspace in a file named RECTANGLES. If you already have a file of that name, the old one is deleted. The LOAD command takes a file name as input and reads the procedure definitions from that file into workspace.²

Notice that the file names given as inputs to SAVE and LOAD must be preceded by quotation marks.³

The CATALOG command lists all the files on the disk. Logo files will be listed with the characters .LOGO appended to the name. For example, the file created by

```
SAVE "RECTANGLES
```

is listed in the catalog as RECTANGLES.LOGO. You should not include the .LOGO part of the name when you use the LOAD or SAVE commands.

To remove a file from the disk, you use the ERASEFILE command, which takes as input the name of the file to be erased. Be careful not to erase a file unless you are sure that you don't need it anymore!

Keep in mind that the name you give your file has *no relation* to the names of the procedures that will be saved in the file. Each time you save a file, *all* procedures in the workspace are included.⁴

² Logo filing makes use of the same memory area as for drawing pictures and editing procedures. Issuing any filing command while in draw mode first moves you to nodraw mode. Compare note 7 in Chapter 1.

³ This is because the file names are considered to be character string data objects, known in Logo as *words*. We will see in section 5.3 that words are quoted in this way when they are typed in to Logo.

⁴ This sometimes causes confusion with beginners. For example, if someone has procedures BOX and HOUSE, they write both SAVE "BOX and SAVE "HOUSE, thinking that a separate file is need for each procedure. The result is that they end up with two files, one called BOX and one called HOUSE, and each file contains *both* procedures. In general, you should name your files with a name that describes the *group* of procedures being saved.

When working on large projects, it is a good idea to save your workspace periodically in a file. Also, as you continue to make modifications, you should keep on disk the last two or three versions of your project. A technique for doing this is to include a version number as part of the file name. For example, the first time you save your RECTANGLES workspace, save it as

SAVE "RECTANGLES1

After you have made modifications, save the updated version as

SAVE "RECTANGLES2

and so on. You can use ERASEFILE to get rid of unneeded versions. As you work on the project, it's a good idea to always keep around the previous two or three versions, just in case you mistakenly save a bad version on disk.

4.2.2. More advanced uses of the file system

Although LOAD and SAVE are almost always used to save and restore complete workspaces, it is also convenient to be able to manipulate files in other ways. For example, suppose you want to merge the procedure definitions in two files to create a larger file. You can do this as follows:

1. Starting with an empty workspace, LOAD the first file.
2. LOAD the second file. Now your workspace contains the definitions in both files.
3. SAVE your workspace as the new, combined file.

As another example, suppose you want to delete a few procedure definitions from a file. To do this, LOAD the file, ERASE the unwanted definitions, and SAVE the new workspace.

By combining the two methods above, you can perform more complicated operations, such as splitting a file into two parts, or transferring a few procedure definitions from one file to another.

Another way to achieve flexibility in managing files is to make use of buried packages. (See section 4.1.3.) When you SAVE your workspace, the contents of buried packages are not included in the file that is written on the disk. It is also possible to LOAD a file directly into a package, by calling LOAD with an additional input. For instance,

LOAD "MYFILE "PROGRAM.AIDS

will load the contents of MYFILE into the PROGRAM.AIDS package. If this package is declared to be buried, the newly added material will be buried as well.

4.3. Obtaining Hard Copy

The Apple Logo implementation contains facilities for directing output to printers. This enables you to make listings of files and procedures.

The Logo command .PRINTER takes a number n as input and directs text that would normally be shown on the screen to the device in slot n . For example, if you have a printer plugged into slot 3, then you can obtain a hard copy listing of the procedure POLY by typing

```
.PRINTER 3
PO "POLY
```

Calling .PRINTER with an input of 0 restores the normal mode of printing on the screen. If the input n is in the range 9 through 15, then $n - 8$ is taken to be the slot number of the device, and the information is also printed on the screen.

4.4. Aids for Debugging

One of the main features of Logo as a computer language for education is that students *design* and *write* programs as well as using them. *Debugging* a program is a crucial part of the programming process.

In order to help with debugging programs, Logo allows you to pause execution of a program to perform various debugging activities (for example to examine the intermediate values of variables at some point in a program) and then resume running the program. This is done with the Logo PAUSE command.

As an example of how PAUSE is used, consider the FLAG and RECTANGLE procedures that we introduced in section 2.1.2, where we have added a new first line to RECTANGLE which instructs Logo to pause:

```
TO RECTANGLE :HEIGHT :LENGTH
PAUSE
FORWARD :HEIGHT
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :HEIGHT
RIGHT 90
FORWARD :LENGTH
RIGHT 90
END
```

```

TO FLAG :HEIGHT
FORWARD :HEIGHT
RECTANGLE (:HEIGHT / 2) :HEIGHT
BACK :HEIGHT
END

```

If you now run FLAG 50, Logo will enter the RECTANGLE procedure and type

```

PAUSING... IN RECTANGLE:
PAUSE
RECTANGLE?

```

Logo first types a message, together with the line that contained the PAUSE instruction. The prompt RECTANGLE? indicates that you are now typing commands within the RECTANGLE procedure. You are free to type and execute *any* Logo command just as if you were at top level. The big difference is that now the *variable names* you use will refer to names in the *local library* of the procedure in which you paused. In the current example, the local libraries for FLAG and RECTANGLE are as shown in figure 2.5 on page 27, so that we could examine RECTANGLE's private variables:

```

FLAG 50
PAUSING... IN RECTANGLE
PAUSE
RECTANGLE? PRINT :HEIGHT
25

```

The CONTINUE command (abbreviated CO) executed from within a PAUSE causes execution to resume at the line following the PAUSE.

To abort a PAUSE and return to top level, you can type THROW "TOPLEVEL.

Besides using the PAUSE command explicitly, you can make Logo pause by typing CTRL-Z, which is like CTRL-G, except that it causes Logo to pause rather than return to top level.



CHAPTER 5

Numbers, Words, and Lists

In the previous chapters, we used turtle geometry to introduce the basic techniques for writing Logo procedures. We now move away from graphics to discuss Logo programs that work with “data.” Like most computer languages, Logo provides operations for manipulating numbers and character strings, which in Logo are called *words*. One significant difference between Logo and other simple programming languages is that Logo also provides the ability to combine data into structures called *lists*. This chapter introduces these three kinds of data objects—numbers, words, and lists—together with simple programs that manipulate them. The most important concept in working with Logo data is the notion of a procedure that *outputs a value*. This is introduced in section 5.2 below. We also discuss the use of Logo variables for naming data, and give a more complete explanation of testing and conditionals than the one provided in section 2.2.2. The material presented here provides enough background to complete many programming projects such as the ones described in Chapter 6.

5.1. Numbers and Arithmetic

We have already seen examples of using numbers in turtle programs. Logo provides the basic arithmetic operations of addition, subtraction, multiplication, and division, denoted by $+$, $-$, $*$, and $/$, respectively. In combining arithmetic operations, multiplications and divisions are performed before additions and subtractions unless you use parentheses to make the grouping explicit. Here are some examples.

```
PRINT 3 + 2 * 5
13
```

```
PRINT (3 + 2) * 5
25
```

The result of an operation is normally used as an input to a command like PRINT or FORWARD, or a procedure like POLY. If you do not supply a command to accept the result of an operation Logo will print an error message:

$(3 + 2) * 5$

I DON'T KNOW WHAT TO DO WITH 25

We have already seen that Logo also allows numbers that include a decimal part. These are often called *real numbers* in computer jargon. Here are some examples of real numbers:

4.56	3.0	.0075	0.0075
12.345	3.	-2.59	-.123

Integers and real numbers may be mixed freely in arithmetic operations, for example:

```
PRINT 3.96 - 12
-8.04
```

Adding, subtracting, or multiplying two integers always yields an integer. Any operation that combines integer and real numbers always yields a real number. Also, the division operation, /, always yields a real number as answer, even when the result is a whole number.

```
PRINT 5 / 2
2.5
PRINT 10 / 5
2.
```

Logo can work with integers between -2^{31} and 2^{31} (i.e., within the range $\pm 2,147,483,648$). If you do an integer computation that exceeds this range (e.g., multiplying 2^{31} by 2^{31}) the computation will be done with real numbers. Logo's real numbers and real-number operations are accurate to 7 significant figures and must have absolute value between (approximately) 10^{38} and 10^{-38} .

In general, you need not worry about whether you are performing operations with integers or real numbers. Bear in mind, however, that the results of arithmetic operations on real numbers are accurate only to 7 decimal places, whereas integer operations are always exact. In addition to the basic operations, Apple Logo includes a few mathematical functions such as sine (SIN), cosine (COS), and square root (SQRT). Section 10.2 gives a complete list of the numeric operations included in Logo.

5.1.1. Exponential notation

Logo also allows you to input real numbers in so-called *exponential notation*. Here are some examples:

3E4	5N2	-4E4	-4N5
3.4E4	4.56N3	-2.5925E5	-259.3N7

In general, exponential notation is of the form:

{number} {E or N} {exponent}

where {number} (also called the *mantissa*) is either an integer or a real number written in standard decimal form and {exponent} is a positive integer. If the letter is E, then the resulting real number is interpreted as {mantissa} $\times 10^{\text{exponent}}$. If the letter is N, then the resulting real number is interpreted as {mantissa} $\times 10^{-\text{exponent}}$. For example, 5.2E3 is the same as 5200.0 and 5.2N3 is the same as .0052. The following are not in correct form:

3E5.1 (the exponent is not an integer)
 2.7E - 1 (use N for 10 to a negative power)
 10 E1 (there should be no space between the 10 and the E)

Although Logo can interpret real numbers that are input in either format, Logo normally prints real numbers in the usual decimal format. If the number is too large or too small to be represented with 7 digits and a decimal point, the number is printed using exponential notation.¹

5.1.2. Integer operations

To convert a real number to an integer, you use the ROUND primitive, which outputs the nearest integer to its input:

ROUND 2.4	outputs	2	ROUND 2.6	outputs	3
ROUND -2.4	outputs	-2	ROUND -2.6	outputs	-3

If the real number is out of the range that can be converted to an integer, Logo signals an error.

An alternative way to convert a real number to an integer is to use the INT operation. This is similar to ROUND, except that it truncates its result by removing any fractional part. Compare the following with the examples of ROUND shown above:

INT 2.4	outputs	2	INT 2.6	outputs	2
INT -2.4	outputs	-2	INT -2.6	outputs	-2

For integer division, you use QUOTIENT, which outputs the integer quotient of its inputs:

¹ A number typed into Logo may therefore not always be typed back in the same format. In addition, numbers used in procedure lines may have their format changed when the procedure is stored in a file and then read back into Logo.

```
PRINT QUOTIENT 5 3
```

```
1
```

REMAINDER gives the remainder of the integer division:

```
PRINT REMAINDER 5 3
```

```
2
```

If QUOTIENT or REMAINDER is given a real number as input, the input is first truncated (using INT) to produce an integer, and the operation is applied to the truncated result:

```
PRINT QUOTIENT 5.01 2.9
```

```
1
```

Random numbers

RANDOM is an integer operation that takes a positive integer n as input and outputs an integer chosen at random between 0 and $n - 1$, inclusive. Thus RANDOM 2 will output either 0 or 1, RANDOM 1000 will output an integer between 0 and 999, and so on.

Each time the Logo system is started, successive calls to RANDOM with identical inputs yield the identical sequence of random numbers. This is useful for debugging programs that use RANDOM. In addition, you can reset the random number generator without restarting Logo, by executing the command RERANDOM.

5.2. Outputs

Using the arithmetic operations presented above, you can write procedures that manipulate numbers. For example,

```
TO PSQUARE :X
```

```
PRINT :X * :X
```

```
END
```

prints the square of its input, and

```
TO PAVERAGE :X :Y
```

```
PRINT (:X + :Y) / 2
```

```
END
```

prints the average of its two inputs:

```
PSQUARE 100
```

```
10000
```

```
PAVERAGE 1 2
```

```
1.5
```

These procedures may be instructive, but they are not very useful. PSQUARE, for example, just prints the square of its input. Having computed the square, there is nothing more you can do with it. Yet the whole power of the procedure concept is that you should be able to use procedures as *building blocks* in defining more complex procedures. You can make complex turtle programs by combining the designs drawn by simple procedures. But there is no way to combine PSQUARE and PAVERAGE to obtain, for instance, the square of the average of two numbers.

What is needed is some way for a procedure not only to compute some result, but also to make that result accessible to other procedures. In Logo, this is accomplished by the OUTPUT command. To see how it works, compare the PSQUARE procedure above with the following:

```
TO SQUARE :X
  OUTPUT :X * :X
END
```

When SQUARE runs, it returns its result as an *output* that is used as an input to whatever command called SQUARE. For example, you can type

```
PRINT SQUARE 3
9
```

in which case the output of SQUARE is passed to PRINT to be printed. More significantly, you could type

```
PRINT (SQUARE 3) + (SQUARE 4)
25
```

Here SQUARE is called twice, and the results are combined by + before being passed to PRINT.

You can do the same thing with computing averages by defining a procedure:

```
TO AVERAGE :X :Y
  OUTPUT (:X + :Y) / 2
END
```

The OUTPUT command is just what is needed to combine operations. For instance, you can find the square of the average of two numbers:

```
PRINT SQUARE (AVERAGE 5 6)
30.25
```

or the average of the squares:

```
PRINT AVERAGE (SQUARE 5) (SQUARE 6)
30.5
```

Alternatively, you can define a procedure to return this value to be used in further processing:

```
TO AVERAGE.OF.SQUARES :X :Y
  OUTPUT AVERAGE (SQUARE :X) (SQUARE :Y)
END
```

As with any procedure, once you have defined a procedure that outputs some result, that procedure becomes part of Logo's working vocabulary and can be used just as if it were a primitive command. For instance, Logo has no primitive absolute value function. But if you define one:

```
TO ABS :X
  IF :X < 0 [OUTPUT (- :X)]
  OUTPUT :X
END
```

then you can use this ABS operation in performing further computations.

When a procedure executes an OUTPUT instruction, it returns the indicated output to the procedure that called it, and no further commands within the procedure are executed. Thus, for example, precisely one of the two OUTPUT instructions in ABS will be executed each time ABS is called.

To help you visualize outputs, figure 5.1 shows a diagram, similar to the diagrams in section 2.1.2, for the procedure calls involved in executing the command line

```
PRINT SQUARE (AVERAGE 5 6)
```

SQUARE and AVERAGE each have a private variable X, but since these are in different private libraries, there is no conflict.

As shown in the diagram, you can regard inputs and outputs as communication channels between procedures. If procedure A calls procedure B then A can use inputs to communicate values to B. B's output enables it to communicate values back to A.

A very common error in Logo programming is to attempt to make a procedure output without using the OUTPUT instruction. For example, you might attempt to define AVERAGE as

```
TO AVERAGE :X :Y
  (:X + :Y) / 2
END
```

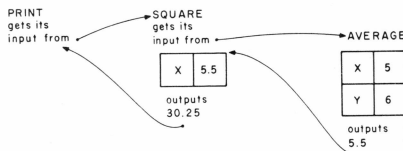


Figure 5.1: Procedure calls in executing PRINT SQUARE (AVERAGE 5 6).

Calling this procedure, say

AVERAGE 5 6

would result in the error message

I DON'T KNOW WHAT TO DO WITH 5.5

when the procedure was executed. In general, you should say what Logo is supposed to do with generated values, for example, print them, output them, or whatever.

5.3. Words

In Logo, strings of characters are called *words*. Logo provides operations for manipulating words: combining words into longer words and breaking words into parts. As with numbers, words may be passed among procedures as inputs and outputs.

To indicate a word in Logo, you type the character string prefixed by a quotation mark, as in:

```
PRINT "WHOOPIE
WHOOPIE
```

Notice that (unlike the rule in English) the quotation mark goes only at the beginning of the word. Beware that if you put a quotation mark at the end of a word, that quotation mark will be taken to be part of the word:

```
PRINT "A"
A"
```

Logo provides the following operations for extracting parts of words:

FIRST	Outputs the first character of its word input.
LAST	Outputs the last character.
BUTFIRST	Abbreviated BF. Outputs a word containing all but the first character.
BUTLAST	Abbreviated BL. Outputs a word containing all but the last character.

Here are some examples:

```
PRINT FIRST "ABCD
A
PRINT BUTFIRST "ABCD
BCD
PRINT LAST BUTLAST "ABCD
C
PRINT BUTFIRST "A
(blank line)
```


In the third example, the thing that is printed is the **LAST** of the **BUTLAST** of **ABCD** which is the **LAST** of **ABC** which is **C**.

Note that in the last example **BUTLAST** outputs all but the first character of the one-character word **A**. The result is a word containing *no* characters. This no-character word is called the *empty word*. According to the rule for inputting words, you can specify the empty word by typing a quotation mark followed by *no* characters:

```
PRINT "  
(blank line)
```

Attempting to extract **FIRST**, **BUTFIRST**, **LAST**, or **BUTLAST** of the empty word causes Logo to signal an error.

For constructing larger words from smaller ones, Logo provides the **WORD** operation. This takes two words as inputs and combines them to form a single word:

```
PRINT WORD "NOW "HERE  
NOWHERE
```

Numbers are considered to be Logo words, and all word-manipulating operations work on numbers:

```
PRINT FIRST 4567  
4  
PRINT BUTFIRST 4567  
567  
PRINT (WORD 12 34) + (WORD 56 78)  
6912
```

Sample procedures that use words

The following recursive procedure is a word analogue of the **COUNTDOWN** procedure on page 36:

```
TO TRIANGLE :WORD  
IF :WORD = " [STOP]  
PRINT :WORD  
TRIANGLE BUTFIRST :WORD  
END
```

```
TRIANGLE "LOLLIPOP  
LOLLIPOP  
OLLIPOP  
LLIPOP  
LIPOP  
IPOP  
POP  
OP  
P
```

Whereas COUNTDOWN reduced a number to smaller numbers by successively subtracting 1, TRIANGLE reduces a word to smaller words by successively removing the first character. The process stops when the word has been reduced to the empty word.

TRIANGLE illustrates the use of words as inputs to procedures. As an example of words as outputs, consider the simple procedure DOUBLE, which takes a word as input and outputs the word concatenated with itself:

```
TO DOUBLE :X
OUTPUT WORD :X :X
END

PRINT DOUBLE "BOOM
BOOMBOOM
```

Observe the importance of using OUTPUT: you can operate on a word using DOUBLE and use the result as an input to other operations:

```
PRINT DOUBLE DOUBLE "BOOM
BOOMBOOMBOOMBOOM

TRIANGLE DOUBLE "ABC
ABCABC
BCABC
CABC
ABC
BC
C
```

5.4. Lists

Many languages force the programmer to work with text in terms of character strings. A long text must be viewed as a long character string that is manipulated on a character-by-character basis. One of the advantages of Logo is that it allows you to manipulate sequences of words on a *word-by-word* basis. In Logo, a sequence of words is called a *list*. A list may be indicated by giving the words in the list separated by spaces and enclosed in square brackets:²

```
PRINT [THIS IS A LIST]
THIS IS A LIST
```

² Logo lists are used not only for making sequences of words, but also for creating data structures in general. See section 9.1. Remember to delimit lists with square brackets [] and not parentheses ().

Notice that the words in the list are not quoted and that the surrounding brackets are not printed. The spaces between the words serve only to delimit the words. Extra spaces are ignored:

```
PRINT [EXTRA      SPACES]
EXTRA SPACES
```

The Logo operations FIRST, LAST, BUTFIRST, and BUTLAST that we introduced for use with words also operate on lists. When used with lists, these operations pick out the first or last word of the list, rather than the first or last character, as they do with words.

```
PRINT FIRST [THIS IS A LIST]
THIS
```

```
PRINT FIRST BUTFIRST [THIS IS A LIST]
IS
```

```
PRINT BUTLAST [THIS IS STILL A LIST]
THIS IS STILL A
```

```
PRINT BUTFIRST [THIS]
(blank line)
```

Note that in the last example, taking all but the first word of a list that has only one word produces a list containing *no* words, called the *empty list*. It can be typed into Logo as []. The empty list is analogous to the empty word, but they are not the same thing. This is a special case of the general rule that a list in Logo is never considered equal to a word. Another case of the same rule is that a word is not considered equal to a list that contains that single word, even though Logo prints these in the same way:

```
PRINT "BUBBLE
BUBBLE
```

```
PRINT [BUBBLE]
BUBBLE
```

```
PRINT "BUBBLE = [BUBBLE]
FALSE
```

SENTENCE is the operation for putting lists together, analogous to WORD for words. SENTENCE (abbreviated SE) takes words or lists as inputs and assembles these into one list:

```
PRINT SENTENCE [THIS IS] [HOW SENTENCE WORKS]
THIS IS HOW SENTENCE WORKS
```

```
PRINT SENTENCE "THIS [IS TOO]
THIS IS TOO
```

```
PRINT SENTENCE "THIS "ALSO
THIS ALSO
```

Sample procedures that use lists

Here are the list procedures analogous to the word procedures TRIANGLE and DOUBLE of section 5.3:

```
TO TRIANGLE.LIST :X
IF :X = [ ] [STOP]
PRINT :X
TRIANGLE.LIST BUTFIRST :X
END
```

```
TO DOUBLE.LIST :X
OUTPUT SENTENCE :X :X
END
```

```
TRIANGLE.LIST [THIS IS A LIST]
THIS IS A LIST
IS A LIST
A LIST
LIST
```

```
PRINT DOUBLE.LIST [HUP 2 3 4]
HUP 2 3 4 HUP 2 3 4
```

```
TRIANGLE.LIST DOUBLE.LIST [DING DONG]
DING DONG DING DONG
DONG DING DONG
DING DONG
DONG
```

The main thing to observe in these examples is that lists, like numbers and words, can be passed between procedures as inputs and outputs.

The following list procedures make use of the Logo command READLIST (abbreviated RL), which makes it easy to write interactive programs using lists. READLIST waits for you to type in a line (terminated by RETURN) and outputs the typed-in line as a list.

```

TO BOAST
PRINT [WHO'S THE GREATEST?]
IF READLIST = [ME] [PRINT [OF COURSE!]] STOP]
PRINT [NO, TRY AGAIN]
BOAST
END

```

```

BOAST
WHO'S THE GREATEST?
MIGHTY MOUSE
NO, TRY AGAIN
WHO'S THE GREATEST?
ME
OF COURSE!

```

Bear in mind that READLIST always outputs a list. If you type in a single word, the output of READLIST will be a list containing that one word. If you just press RETURN without typing anything, the output of READLIST will be the empty list.

Here's another example:

```

TO CHAT
PRINT [WHAT'S YOUR NAME?]
PRINT SENTENCE [HELLO] READLIST
PRINT [TYPE SOMETHING YOU LIKE]
PRINT SENTENCE [I'M GLAD YOU LIKE] READLIST
END

```

Notice how the second line of the procedure is constructed: the list being PRINTed is a SENTENCE of two things—the list [HELLO] and the list output by READLIST.

```

CHAT
WHAT'S YOUR NAME?
LUCY
HELLO LUCY
TYPE SOMETHING YOU LIKE
PICKLE JELLO
I'M GLAD YOU LIKE PICKLE JELLO

```

5.5. Naming

We have so far seen two different kinds of *naming* in Logo programs: the use of names to refer to inputs to procedures, and the idea of naming procedures themselves. But we have not yet seen Logo programs in which names are given freely to data.

The Logo command used to name things is MAKE. Consider the following:

```
MAKE "NUMBER 5
PRINT :NUMBER
5
```

In the first line you tell Logo that you are going to call the number 5 by the name NUMBER. The first input to MAKE is the *name* and the second input is the *thing* you are naming. The effect of the command is to establish a relationship between the word NUMBER and the number 5. We express this by saying that "5 is the thing associated with NUMBER." In the line

```
PRINT :NUMBER
```

you can see how : recovers the thing associated with the name, just as it recovers the value associated with an input to a procedure. Here are more examples:

```
MAKE "COLR "YELLOW
PRINT "COLR
COLR
```

```
PRINT :COLR
YELLOW
```

```
MAKE "SLOGAN [I LOVE BANANAS]
PRINT :SLOGAN
I LOVE BANANAS
```

```
PRINT SENTENCE (BUTLAST :SLOGAN) :COLR
I LOVE YELLOW
```

In these examples, and in most programs, the name is specified as a literal, quoted word. This is not the only possibility:

```
MAKE (WORD "PART 1) [DO MI SOL]
PRINT :PART1
DO MI SOL
```

Here is a tricky example:

```
MAKE "FLOWER "ROSE
PRINT :FLOWER
ROSE
```

```
MAKE :FLOWER [IS A ROSE IS A ROSE]
PRINT :FLOWER
ROSE
```

```
PRINT :ROSE
IS A ROSE IS A ROSE
```

In the third command line, the name associated with [IS A ROSE IS A ROSE] is not the literal word FLOWER, but rather the *thing* associated with FLOWER, that is, the word ROSE. Therefore

```
MAKE :FLOWER {something}
```

has the same effect as

```
MAKE "ROSE {something}
```

The Logo function THING returns the thing associated with its input. The use of : is actually an abbreviation for THING in the case where the input to THING is a quoted literal word. But THING can be used in more general circumstances:

```
MAKE "NAME1 [JOHN Q. CITIZEN]
```

```
PRINT :NAME1
```

```
JOHN Q. CITIZEN
```

```
PRINT THING "NAME1
```

```
JOHN Q. CITIZEN
```

```
PRINT THING (WORD "NA "ME1 )
```

```
JOHN Q. CITIZEN
```

```
PRINT THING (FIRST [NAME1 PLACE1])
```

```
JOHN Q. CITIZEN
```

There is also the Logo predicate NAMEP, which takes a word as input and outputs TRUE if the word is the name of a Logo thing.

5.5.1. Local and global names

In section 2.1.2 we saw that the names of inputs are *private* to the procedures using them. Different procedures reference names in different private libraries, and two procedures may use the same names for different purposes without any conflict. The same holds true if the procedure uses the MAKE command to change the value associated with some input name. This is illustrated in the following example:

```
TO DEMO :X
```

```
PRINT :X
```

```
CHANGE :X
```

```
PRINT :X
```

```
END
```

```
TO CHANGE :X
```

```
MAKE "X :X + 1
```

```
PRINT :X
```

```
END
```

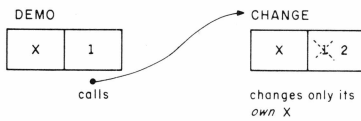


Figure 5.2: Private libraries for DEMO and CHANGE.

DEMO 1

```
1      (printed in DEMO)
2      (printed in CHANGE)
1      (printed in DEMO)
```

The important point to notice is that when the value of *X* is printed in DEMO the second time, it is still 1 even though CHANGE “changed” *X* to 2. The reason is that DEMO and CHANGE each have their own meaning for *X* in different private libraries, as shown in figure 5.2. When CHANGE uses the MAKE statement it changes *its* *X*, but not DEMO’s.

When you use a MAKE statement at command level, you are also associating a value with a name in some library. But this is not a library associated with any procedure. Rather it is a library associated with the command level. Definitions in this library are sometimes called *global variables*. Just as the private libraries of two procedures are distinct, names in procedure libraries will not conflict with names in the global library. Compare the following example to the one above.

```
MAKE "X 1
CHANGE :X
2
PRINT :X
1
```

5.5.2. Free variables

One of the reasons that procedures are so important is that they provide a way to design complex programs in small pieces. But whenever you design something by breaking it into pieces, you eventually have to deal with the issue of how these pieces can interact. The importance of the private library mechanism is that it guarantees that the names used by different procedures will refer to different things and hence that the *only* way procedures can interact is through inputs and outputs. This guarantee provides a good handle on controlling the complexity of the entire program.

Sometimes, however, it is convenient for procedures to interact other than through inputs and outputs. For example, if the computation performed by a procedure depends on a large number of parameters, it may be cumbersome to specify them all as inputs each time the procedure is called. Again, using only inputs and outputs to pass information may require passing “superfluous” inputs through many levels of nested procedures until they reach the procedure that actually needs them. For these reasons it is useful to be able to have the computation performed by a procedure depend not only on the information provided *explicitly* by the inputs, but also on information that is *implicit* to the *context* in which the proce-

cedure is used.

Consider the following procedure:

```
TO NEW.PRICE :P
  OUTPUT :P + :OVERHEAD
END
```

Suppose you would like to be able to use this procedure in such a way that the price computed depends on some OVERHEAD that is obtained from the context in which the procedure is used. For example,

```
MAKE "OVERHEAD 50
PRINT NEW.PRICE 100
150
MAKE "OVERHEAD 25
PRINT NEW.PRICE 100
125
```

You might also want to have the context determined by a procedure, as in

```
TO TRY :OVERHEAD
  PRINT NEW.PRICE 100
  PRINT NEW.PRICE 200
END

TRY 100
200
300

TRY 50
150
250
```

The name OVERHEAD in the NEW.PRICE procedure is what is technically known as a *free variable*. A free variable is a name that is used in a procedure, but not as a name for an input. As the NEW.PRICE example shows, Logo procedures can have free variables. The presence of free variables leads to the following rule for finding the value associated with a name in a Logo procedure:

- If the name is one of the names of the inputs to the procedure, the value can be found in the procedure's private library.
- Otherwise, see if the name is in the library of the procedure that *called* the current procedure.

- Otherwise, see if the name is one of the names in the procedure that called *that* procedure, and so on, all the way through to the global library.

Free variables provide a powerful mechanism for passing information between procedures. But their indiscriminate use leads to obscure programs and may result in intractable program bugs. This is especially true if you use MAKE to change the value of a free variable, since the actual variable affected may appear arbitrarily far back in the nest of procedure calls.

5.6. Conditional Expressions and Predicates

We saw in section 2.2.2 the use of conditional expressions IF... in Logo programs. This section provides more information about conditional expressions.

The general form of an IF statement is

IF some condition [action] [alternative]

For example, the following procedure tells whether a number is positive or negative:

```
TO SIGN :N
IF :N < 0 [OUTPUT "NEGATIVE] [OUTPUT "POSITIVE]
END
```

```
PRINT SIGN 57
POSITIVE
PRINT SIGN (10 - 20)
NEGATIVE
```

IF expressions are often confusing for beginning programmers, due to the need to work with a single statement that specifies both a test and the actions to be taken depending on the outcome of the test. Logo therefore includes another form of conditional that separates the testing from the actions. This form is TEST . . . IFTRUE . . . IFFALSE. The TEST used in a procedure checks some condition. Subsequent procedure lines that begin with IFTRUE and IFFALSE are executed or not depending on the result of the TEST. Here's another way to write the SIGN procedure using TEST:

```
TO SIGN :N
TEST :N < 0
IFTRUE [OUTPUT "NEGATIVE]
IFFALSE [OUTPUT "POSITIVE]
END
```

A procedure can include more than one TEST, and any IFTRUE or IFFALSE statements always refer to the most recent TEST. Also, the result of a TEST is kept private within a procedure, so the use of IFTRUE and IFFALSE within a procedure is not affected by any TESTs performed in a subprocedure.

Predicates: the words TRUE and FALSE

The conditions checked by IF and TEST are known as *predicates*. We already introduced the Logo predicates >, <, and = for working with numbers. Section 10.6 gives a complete list of the predicates built in to Logo. It is also easy to define new predicates, because a predicate in Logo is nothing more than a procedure that outputs either the word TRUE or the word FALSE. For instance, you can transform the SIGN procedure given above into a predicate that outputs FALSE if the input is less than 0 and TRUE otherwise:³

```
TO POSITIVE? :X
IF :X < 0 [OUTPUT "FALSE] [OUTPUT "TRUE]
END
```

As another example, the following predicate takes a word as input and tests whether it begins with a vowel:

```
TO BEGINS.WITH.VOWEL? :X
IF (FIRST :X) = "A [OUTPUT "TRUE]
IF (FIRST :X) = "E [OUTPUT "TRUE]
IF (FIRST :X) = "I [OUTPUT "TRUE]
IF (FIRST :X) = "O [OUTPUT "TRUE]
IF (FIRST :X) = "U [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

Once a predicate has been defined, it can be used with IF or TEST just as if it were one of the predicates built in to Logo. Here is a procedure that adds "a" or "an" to a word, as appropriate:⁴

³ It is a good programming habit to name predicates with names that end with a question mark. As far as the Logo system is concerned, though, the question mark has no special significance. It is treated as an ordinary character. Logo's built-in predicates tend to have names that end in P (for "predicate").

⁴ Later on, when we see how to take advantage of Logo lists as data structures, we will learn more flexible ways of computing functions like BEGINS.WITH.VOWEL?. Compare the alternative version of the ADD.A.OR.AN procedure on page 166.

```

TO ADD.A.OR.AN :X
TEST BEGINS.WITH.VOWEL? :X
IFTRUE [OUTPUT SENTENCE "AN :X]
IFFALSE [OUTPUT SENTENCE "A :X]
END

```

```

PRINT ADD.A.OR.AN "COMPUTER
A COMPUTER
PRINT ADD.A.OR.AN "APPLE
AN APPLE

```

You can regard IF and TEST as operations that take an input that must be either the word TRUE or the word FALSE. In fact, the primitive predicates built in to Logo are themselves also operations that output TRUE or FALSE:

```

PRINT 3 > 5
FALSE
PRINT "XYZ = "XYZ
TRUE

```

For combining predicates, Logo includes the operation AND, which takes two inputs that must be either TRUE or FALSE, and outputs TRUE if both inputs are TRUE and FALSE otherwise. There is also OR, which outputs TRUE if at least one of its inputs is TRUE, and NOT, which outputs TRUE if its input is FALSE, and FALSE if its input is true.

```

PRINT AND (1 < 2) (2 > 3)
FALSE
PRINT OR (1 < 2) (2 > 3)
TRUE
PRINT NOT (2 + 2 = 4)
FALSE

```

For example, here are three equivalent ways of writing a predicate BETWEEN?, which tests whether a specified number is in a given range:

```

TO BETWEEN? :X :LOW :HIGH
IF: X < :LOW [OUTPUT "FALSE]
IF: X > :HIGH [OUTPUT "FALSE]
OUTPUT "TRUE
END

```

```

TO BETWEEN? :X :LOW :HIGH
IF OR (:X < :LOW) (:X > :HIGH) [OUTPUT "FALSE]
OUTPUT "TRUE
END

```

Type as one line. See page 9.—

```
TO BETWEEN? :X :LOW :HIGH
IF AND (NOT (:X < :LOW)) (NOT (:X > :HIGH))
  [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

AND and OR are themselves predicates that output TRUE or FALSE. This means that the second two versions of BETWEEN? can also be written in another way, in which the TRUE or FALSE output by AND and OR is output directly to the procedure that calls BETWEEN?:

```
TO BETWEEN? :X :LOW :HIGH
OUTPUT AND (NOT (:X < :LOW)) (NOT (:X > :HIGH))
END
```

```
TO BETWEEN? :X :LOW :HIGH
OUTPUT NOT OR (:X < :LOW) (:X > :HIGH)
END
```

5.7 Details of Logo Syntax

This section collects some information about how Logo interprets the command lines that you type to it. This includes such information as where to include spaces and parentheses in command lines, and how Logo groups sequences of commands. More detailed information about how Logo “parses” command lines is provided in the technical manual, Appendix I, Parsing.

5.7.1. How Logo separates lines into words

Any Logo line is interpreted as a sequence of words. In general, you must separate words by spaces. For example, if you mean to type

```
FORWARD 100
```

and instead type

```
FORWARD100
```

Logo will respond with the error message

```
I DON'T KNOW HOW TO FORWARD100
```

because it will interpret FORWARD100 as a single word and look for a procedure with that name.⁵ As a general rule it is a good idea to type each line with spaces between the different elements, for example,

⁵ Of course, you may have actually defined a procedure whose name was FORWARD100, in which case Logo would run that procedure.

```
PRINT ( 3 + 4 ) * 5
35
```

Logo does, however, understand that parentheses and arithmetic operators are normally meant to break words, so

```
PRINT (3 + 4) * 5
35
```

works, too. In fact, if you define a procedure that includes a line such as the previous one, and later print out the procedure, you will find that Logo has inserted spaces into the line.

5.7.2. Using parentheses

We have already seen some complex Logo expressions, for example, the following line from the AVERAGE.OF.SQUARES procedure on page 80:

```
OUTPUT AVERAGE (SQUARE :X) (SQUARE :Y)
```

In this line, the OUTPUT command takes one input, which is the result of AVERAGE. AVERAGE in turn takes two inputs:

```
(SQUARE :X)
```

and

```
(SQUARE :Y)
```

Notice that parentheses perform grouping by enclosing the operation *together with its inputs*. That is, you should write

```
(SQUARE :X)
```

and not

```
SQUARE (:X)
```

to indicate that :X is the input to SQUARE.⁶

In fact, this expression would work perfectly well if you wrote it without any parentheses at all,

```
OUTPUT AVERAGE SQUARE :X SQUARE :Y
```

⁶ This rule can be confusing, since the latter expression is more like SQUARE(X), which is what is used in mathematics or in languages like BASIC or Pascal. Keep in mind that parentheses in Logo are used to indicate *grouping*, and are not special symbols for delimiting the list of inputs to functions.

because when Logo interprets the line it breaks things up according to the following method. The first word it sees is OUTPUT, and this requires one input. So Logo scans the line trying to find that input. The next thing it runs into, though, is AVERAGE, which requires two inputs of its own. So Logo now scans to find two inputs for AVERAGE and runs into SQUARE, which requires one input which logo finds as :X. This completes the inputs to SQUARE, and also completes the first input to AVERAGE. Logo now looks for the second input to AVERAGE, and the next thing it sees is another SQUARE, which requires one input. Logo finds this as :Y. Now SQUARE has its input. This completes both inputs to AVERAGE, which completes the entire input to OUTPUT.

Generalizing this method, you can see that so long as Logo knows how many inputs each function needs, and so long as each function name is a prefix operator (i.e., it is written to the left of its inputs), then you don't need parentheses at all in writing Logo commands. On the other hand, parentheses help considerably in enabling the human eye to see the pattern. So unless you are very practiced, you should not write a complex expression without parentheses for fear of not being able to read it the next day.

The above rule for parsing expressions is modified for infix operators (i.e., operators that are written between their inputs, rather than to the left of them). In a line such as

WORD 3 + 4 7

the 3 is combined with the 4 by + before any unit is assigned as an input to WORD, so the line gets broken up as:

WORD (3 + 4) 7

which gives 77. The general rule is that the infix arithmetic operators +, -, *, and / have higher priority than prefix operators.

The infix predicates >, <, and = have lower priority than prefix operators. So

WORD 3 5 > WORD 2 3

is combined as

(WORD 3 5) > (WORD 2 3)

Logo's rules for parentheses are designed to enable you to write simple expressions without worrying about parentheses. For complex expressions, it is better to use parentheses to avoid confusion.

Commands with a variable number of inputs

Although we haven't mentioned it yet, certain Logo primitive operations can take a variable number of inputs. One of them is SENTENCE, as in the following example:

```
PRINT (SENTENCE [THE BIG] [BAD] [WOLF])
THE BIG BAD WOLF
```

which uses one SENTENCE operation to combine three things into a list. The fact that SENTENCE is combining three things rather than its usual two is indicated by the parentheses grouping SENTENCE together with its inputs. In this way, SENTENCE can take any number of inputs. Other Logo commands that take a variable number of inputs are WORD and PRINT. (When PRINT is called with more than one input, it prints all its inputs on one line, separated by spaces.)

```
PRINT (WORD "ONE "TWO "THREE "FOUR)
ONETWOTHREEFOUR

(PRINT "ONE "TWO "THREE "FOUR )
ONE TWO THREE FOUR
```

The operations AND and OR also can take variable numbers of inputs:

```
PRINT (AND (5 = 3) (5 = 4) (5 = 5))
FALSE

PRINT (OR (5 = 3) (5 = 4) (5 = 5))
TRUE
```

Examples

Here are a few examples illustrating the rules discussed above, including some common errors and their explanations:

```
PRINT "A "B "C
A
I DON'T KNOW WHAT TO DO WITH B
```

The default number of inputs to PRINT is 1, so PRINT prints its input, which is A. Now Logo is faced with the rest of the line, and it runs across the symbol "B." Since there are no outstanding operations that need inputs, Logo complains that there is nothing to do with the B.


```
(PRINT "A "B "C)  
A B C
```

Here parentheses are correctly used to group the three inputs to PRINT.

```
PRINT ("A "B "C)  
TOO MUCH INSIDE ( )'s
```

When you use parentheses to indicate grouping, you should group an operation together with its inputs. The parentheses are being used in this example as they would be used in BASIC, to surround the inputs alone. But Logo always tries to interpret a parenthesized expression as a complete unit, which does not make sense in this case.

CHAPTER 6

Projects Using Numbers, Words, and Lists

This chapter presents for beginning students three open-ended projects using numbers, words, and lists. The first project is a simple arithmetic quiz program similar to the drill and practice computer systems used in some schools. The next project shows how to use lists to write programs that generate “random” sentences. We then reprint a paper by Papert and Solomon [11] that describes a simple game-playing program and discusses ideas about how to involve students in planning and carrying out complex projects.

6.1. Arithmetic Quiz Program

Here’s a simple arithmetic drill and practice program:

```
QUIZ
HOW MUCH IS 37 + 64
101
GOOD
HOW MUCH IS 29 + 46
87
THE ANSWER IS 75
HOW MUCH IS 21 + 11
32
GOOD
and so on.
```

Designing a quiz program that works like this is a good programming project for elementary school students.¹ Here is one of many possible versions:

```
TO QUIZ
MAKE "NUM1 RANDOM 100
MAKE "NUM2 RANDOM 100
MAKE "ANSWER :NUM1 + :NUM2
PRINT (SENTENCE [HOW MUCH IS] :NUM1 [ + ] :NUM2)
MAKE "REPLY READNUMBER
```

¹ And designing such a program is probably a better educational experience than *using* such a program, which is unfortunately much more typical of how computers are currently used in schools.

```

TEST :REPLY = :ANSWER
IFTRUE [PRINT [GOOD]]
IFFALSE [PRINT SENTENCE [THE ANSWER IS] :ANSWER]
QUIZ
END

TO READNUMBER
OUTPUT FIRST READLIST
END

```

You use READNUMBER rather than READLIST directly because READLIST outputs a list. If the user types in a single number, READLIST outputs a list containing that number as its only item. To obtain the number itself, you extract the first item from the list returned by READLIST.²

You can also modify READNUMBER to check that the response is actually a number:

```

TO READNUMBER
MAKE "IN FIRST READLIST
TEST NUMBERP :IN
IFTRUE [OUTPUT :IN]
IFFALSE [PRINT [PLEASE ANSWER WITH A NUMBER]]
IFFALSE [OUTPUT READNUMBER]
END

```

The behavior of QUIZ is now:

```

QUIZ
HOW MUCH IS 6 + 14
BO
PLEASE ANSWER WITH A NUMBER
20
GOOD
etc.

```

Observe that the recursive call in the final line of the READNUMBER procedure makes the procedure keep asking until the user responds with a number.

QUIZ is a good programming project because it has a simple core, yet there are so many extensions and variations. Some of these are as follows:

² Thus, if you set ANSWER to be the list returned by READLIST, ANSWER will never be equal to the sum of NUM1 and NUM2, which is a number. For example, if the user types 7 followed by RETURN, the value returned by request will be the list [7], not the number 7.

- Allowing the user to keep trying a question until getting the correct answer
- Keeping score
- Progressing to harder and harder problems when the score is good
- Giving advice

6.2. Random-Sentence Generators

You can have lots of fun with programs that print random sentences. In designing such programs it is very useful to have as a building block a procedure PICKRANDOM that takes a list as input and outputs an item selected at random from a list, for example:

```
PRINT PICKRANDOM [EENEY MEENEY MINEY MO]
MEENEY
PRINT PICKRANDOM [EENEY MEENEY MINEY MO]
MO
```

PICKRANDOM is not built in to Logo as a primitive, but it can be implemented by using lists and recursion, PICKRANDOM is implemented in terms of Logo primitives ITEM, which outputs the n th item in a given list, and COUNT, which outputs the number of items in a list:

```
TO PICKRANDOM :X
  OUTPUT ITEM (1 + RANDOM (COUNT :X)) :X
END
```

Observe how the inputs to ITEM and RANDOM are chosen. If the COUNT of the list X is n , then

```
RANDOM (COUNT :X)
```

returns a number selected at random between 0 and $n - 1$. You should add 1 to this to produce a random number between 1 and n , which becomes the input to ITEM. For beginning users, you can supply the entire PICKRANDOM procedure as a "black box."

Once you have PICKRANDOM, it is easy to generate simple random sentences of the form {noun} {verb} by picking words at random from lists of nouns and verbs:

```
TO CHATTER
  MAKE "NOUNS [DOGS CATS CHILDREN TIGERS]
  MAKE "VERBS [RUN BITE TALK LAUGH]
  BABBLE
END
```

Type as one line. See page 9.→

```
TO BABBLE
PRINT SENTENCE (PICKRANDOM :NOUNS)
                (PICKRANDOM :VERBS)
```

```
BABBLE
END
```

```
CHATTER
CATS LAUGH
TIGERS TALK
CHILDREN BITE
TIGERS BITE
DOGS TALK
```

You can make the sentence generator more interesting by telling the computer occasionally to ask for a new noun or verb to be typed in, and to add it to the corresponding list. For nouns this can be done with

```
TO LEARN.NOUN
PRINT [TEACH ME A NEW NOUN]
MAKE "NOUNS SENTENCE :NOUNS READLIST
END
```

Observe that this uses the SENTENCE operation to combine the typed in word with the list of current nouns. There is a similar LEARN.VERB procedure for verbs.

Now you can modify BABBLE to ask for a new noun or verb every so often (1 chance in 10):

```
TO BABBLE
IF (RANDOM 10) = 0 [LEARN.NOUN]
IF (RANDOM 10) = 0 [LEARN.VERB]
PRINT SENTENCE (PICKRANDOM :NOUNS)
                (PICKRANDOM :VERBS)
```

Type as one line. See page 9.→

```
BABBLE
END
```

The behavior of the program is now

```
CHATTER
CHILDREN TALK
TIGERS RUN
TEACH ME A NEW VERB
WALK
DOGS RUN
```

CATS BITE
 TEACH ME A NEW NOUN
 BANANAS
 DOGS WALK
 BANANAS BITE

There are many extensions to this project, including making more complex sentences by adding other parts of speech such as adjectives and adverbs, matching singular verbs with singular nouns and plural with plural, and generating random "poetry." Papert [12] describes the experience of one 13-year-old while engaged in such a project:

One day Jenny came in very excited. She had made a discovery. "Now I know why we have nouns and verbs," she said. For many years in school Jenny had been drilled in grammatical categories. She had never understood the differences between nouns and verbs and adverbs. But now it was apparent that her difficulty with grammar was not due to an inability to work with logical categories. It was something else. She had not been able to make any sense of what grammar was about in the sense of what it might be *for* . . . But now, as she tried to get the computer to generate poetry, something remarkable happened. She found herself classifying words into categories, not because she had been told she had to but because she needed to. In order to "teach" her computer to make strings of words that would look like English, she had to "teach" it to choose words of an appropriate class. What she learned about grammar from this experience with a machine was anything but mechanical or routine. Her learning was deep and meaningful. Jenny did more than learn definitions for particular grammatical classes. She understood the general idea that words (like things) can be placed in different groups or sets, and that doing so could work for her. She not only "understood" grammar, she changed her relationship to it.

6.3. Nim: a Game-Playing Program

This section is a slightly modified version of a paper written by Seymour Papert and Cynthia Solomon, which was originally published as an MIT Artificial Intelligence Laboratory Memo [11]. It illustrates some ideas about how to initiate beginning students into the art of planning and writing a program complex enough to be considered a project rather than an exercise on using the language or simple programming ideas.

The project is to write a program to play a simple game (“one-pile Nim” or “21”) as invincibly as possible. The project was developed by Papert and Solomon for a class of seventh-grade students taught during 1968–69 at the Muzzey Junior High School in Lexington, Mass. This was the longest programming project these students had encountered, and the intention was to give them a model of how to go about working under these conditions. To achieve this, the teachers worked very hard to develop a clear organization of sub-goals, which they explained to the class at the beginning of the three-week period devoted to this particular program. You would not expect beginners to find as clear a sub-goal structure as this one; but once they have seen a good example, they are more likely find clear sub-goals in the future for other problems. Thus the primary teaching purpose was to develop the idea of splitting a task into sub-goals. The intent was to provide the students with good models of various ways in which this can be done, and to have them experience the heuristic power of this kind of planning (as opposed to jumping straight into writing programs).

A sub-goal structure can be imposed on a problem in several ways. One way is by “chopping,” that is, by recognizing that the final program has distinct functions that can be performed by separate subprocedures. But this is not the only way. Many heuristic programs can be simplified rather than chopped. We illustrate this by first writing a procedure to play the entire game of Nim, but in a “dumb way.” Once we have done so, we can study the procedure’s performance, decide why it plays badly and strengthen its play. Thus the successive partial solutions to the problem appear as making a procedure progressively “smarter.”

Describing the evolution of the program in this way has the additional benefit of allowing one to make an analogy valuable in two senses: by using themselves as models, students acquire a fertile source of ideas about programming; on the other hand, the experience of debugging programs can have a therapeutic effect in leading them to see their own mistakes as emotionally neutral *bugs* rather than as emotionally charged *errors*.

6.3.1. The sub-goal plan

The key idea for subdivision of the problem is to write a series of programs, each of which is “smarter” than the previous one. The first program knows nothing about the strategy of play. It does not generate moves, but asks each of two human players in turn what move to make. For example, it may act as a scorekeeper, just keeping track of the number of sticks without bothering about whether the move is legal. From scorekeeper the machine can advance to referee. This means that it checks the moves for legality and eventually declares the

game over and announces the winner. After we have a working mechanical referee, we start making a mechanical player. The first version of a player chooses legal, but not necessarily good moves. Indeed, it generates a move randomly, uses its ability as a referee to decide if it is legal, and then either accepts it or generates another random move.

When this works, the child may make the program smarter and smarter by adding features or by writing a completely new version until finally if all goes well—an infallible strategic player is evolved.

A natural form for programs of intermediate smartness is the following: the program has a list of simple situations in which it knows how to play; in other situations it plays randomly. In other words, it plays by the form of strategy used by most children in most strategic games.

In working with a class, a good moment should be seized to prod the students into noting and discussing the analogy between this very simple heuristic program and themselves—particularly, how the program gets to be smarter through more or better knowledge. Seeing the program as a cognitive model is a valuable and exciting experience for the students. They can easily be drawn into discussion about how meaningful such models are. To keep the discussion alive, the teacher should be equipped with arguments and examples to counteract extremist, and so sterile, positions. For example, if the students feel that the program is too simple to be a model of human thinking, the teacher might discuss whether a toy airplane is a useful model of a jet-airplane. Does it work by the same principles? Can you learn about airliners by studying toy models? On the other hand, if a class swings over to the position that there really is no difference, the teacher can ask questions about whether the program could learn by itself without a programmer. If this is too enthusiastically accepted, it is well for the teacher to ask: "How much do you learn without being told?" Ideally, the teacher should merely guide the discussion without having to say any of this. But awareness of such argument will permit more sensitive guiding. An interesting exercise and base for discussion is to have the students study various programs of intermediate smartness, classify their bad moves by degrees of stupidity, and give the program grades (or say why they think doing so is silly!).

The stratification of the project has the good feature of allowing students to find their own levels. A slower child who gets only as far as the random player, nevertheless has the taste of success if his program does what it does well. Tendencies to feel inferior should be counteracted by the teacher's attitude and by the teacher's encouraging individual variations so that no child's final program is a mere subset of a more advanced

one. The teacher's computer culture can be very relevant in this delicate kind of situation. Although the richness of programming permits students to generate many fertile ideas, sensitive filtering by the teacher can enormously improve the achievement-to-frustration ratio.

First steps with the students

A move in Nim consists of taking one, two, or three matchsticks from a given pile. Two players move alternately. The player who takes the last stick wins.

The first step is to see that everyone knows the rules and understands what the first program does, for example, by imitating its function or by writing imaginary scripts. In the course of discussing this the teacher introduces some names so the class can talk about what the program is doing.

Here is an example of a script:

```
THE NUMBER OF STICKS IS 8
JOAN TO PLAY. WHAT'S YOUR MOVE?
2
THE NUMBER OF STICKS IS 6
BILL TO PLAY. WHAT'S YOUR MOVE?
3
THE NUMBER OF STICKS IS 3
JOAN TO PLAY. WHAT'S YOUR MOVE?
3
```

Later in the project the teacher can insist that students consider what happens when a player replies to WHAT'S YOUR MOVE by 5 or COW. In the beginning the teacher should discourage all but the most competent students from worrying about "funny" answers before getting the program to work with normal answers.

Examining the script you see that there must be names for:

- The current number of sticks—say STICKS
- The move—say MOVE
- The next player—say PLAYER
- And, a little more subtle, the other player—say OPPONENT

To be sure that everyone understands, they are asked to fill in these "Logo things" for successive rounds, following the previous script.

Round No.	:STICKS	:PLAYER	:OPPONENT	:MOVE
1	8	JOAN	BILL	2
2	?	?	JOAN	3
3	3	?	?	?

6.3.2. A simple scorekeeper

If this is the first game-playing program, the teacher builds up to it by asking some standard questions:

- What shall we call the procedure? (Let's say NIMPLAY)
- What must NIMPLAY do?
- What must NIMPLAY know?

Possible answers are

- Announce the remaining number of sticks.
- Announce the player to move.
- Get the move and make all the modifications.
- Recurse.

To do this, NIMPLAY must remember :STICKS, :PLAYER, and :OPPONENT from the previous round and get :MOVE by asking for it. The first three things must be passed from one call to NIMPLAY to the next so they should be inputs. On the other hand, :MOVE comes from the human player, so it does not need to be an input. If you look ahead, you may notice that later on, :MOVE will sometimes come from a procedure—that is, when the machine gets to be smart enough to make its own moves. So to keep the door open for changes, the problems of getting :MOVE and using it are separated. The standard way to do this is to plan on a subprocedure—say, called GETMOVE.

Now students can write NIMPLAY:

```

TO NIMPLAY :STICKS :PLAYER :OPPONENT
    When in doubt, have lots of inputs.
    PRINT SENTENCE [THE NUMBER OF STICKS IS] :STICKS
        Announce the number of sticks.
    PRINT SENTENCE :PLAYER
        [TO PLAY. WHAT'S YOUR MOVE?]
    MAKE "NEWSTICKS :STICKS - GETMOVE
        Pretend GETMOVE has already been written.
    NIMPLAY :NEWSTICKS :OPPONENT :PLAYER
        Recursion line. Notice how :PLAYER
        and :OPPONENT are reversed.
END

```

Type as one line. See page 9.—

```

TO GETMOVE
MAKE "MOVE READNUMBER
    See READNUMBER procedure on page 100.
OUTPUT :MOVE
END

```

From scorekeeper to referee

As referee the program has some new tasks:

- To decide whether the game is over
- To declare the winner if it is over
- To make sure that :PLAYER takes 1, 2, or 3 sticks each time

The first two tasks are achieved by adding a test and a stop line to NIMPLAY. For example,

```

TEST :NEWTICKS = 0
IFTRUE [PRINT SENTENCE :PLAYER [IS THE WINNER!]]
IFTRUE [STOP]

```

The third task can be accomplished by giving GETMOVE a “try-again” form, using the MEMBERP predicate, which takes an item and a list as inputs and checks whether the item is in the list.

```

TO GETMOVE
PRINT [YOU MAY TAKE 1, 2, or 3 STICKS]
MAKE "MOVE READNUMBER
TEST MEMBERP :MOVE [1 2 3]
IFFALSE [OUTPUT GETMOVE] If the TEST is FALSE, try again.
OUTPUT :MOVE
END

```

With these changes, NIMPLAY is certainly a referee—but still has some rough edges. For example, when :STICKS is 2, GETMOVE gives permission to take 1, 2, or 3 sticks! And if :PLAYER takes 3 sticks, :STICKS becomes negative, and the game will go on forever, because of a “slip-by” bug. However, we shall leave it as an exercise to remedy these minor failings.

In presenting this section to students, the teacher may want to work through one of the two major modifications with the class and let the students struggle with the other. The slip-by bug we would leave to the class to discover and cure. Those who miss it at this stage will find its presence more obtrusive later. If so, a profitable discussion may develop on the question

of why the bug was not found—perhaps because the human player always makes reasonable moves so that :STICKS never becomes negative even though the machine allows it. Later we shall see that when the machine makes its own moves, it is not so cooperative unless it is told to be.

Examples of individual frills to a referee program are: timing moves, declaring the winner a move or two ahead (!), allowing a player to take a move back, printing a score sheet, giving advice (!), establishing and imposing handicaps (!), and changing the rules.

6.3.3. A mechanical player

How can the machine choose a move? The simplest way is by using PICKRANDOM.³ For example, you could allow GETMOVE to make the choice: if a person is to play, use READNUMBER; if the machine is to play, use PICKRANDOM.⁴ But it has to be told whether the player is human or the computer. So it must have an input.

```
TO GETMOVE :PLAYER
TEST :PLAYER = [COMPUTER]
IFTRUE [MAKE "MOVE PICKRANDOM [1 2 3]]
IFFALSE [PRINT [YOU MAY TAKE 1, 2, OR 3 STICKS]]
IFFALSE [MAKE "MOVE READNUMBER]
.
.
. as before.
```

At this stage the slip-by bug may become serious. One way to kill it is to tell GETMOVE about :STICKS and have it try again if :MOVE comes up greater than :STICKS. To do this you change the title line to:

```
TO GETMOVE :PLAYER :STICKS
```

and add a pair of lines after the two MAKES.

```
TEST :MOVE > :STICKS
IFTRUE [OUTPUT GETMOVE :PLAYER :STICKS]
```

³ The PICKRANDOM procedure (page 101) can be written by the teacher and given to students as a "primitive."

⁴ Notice this anthropomorphism. We find it useful to talk of procedures as agents, of their "state of knowledge," of "telling them," of having them "talk to" one another. And we present this to students as a deliberate metaphor that they may find useful.

Strategic play

The plan for writing the Nim-playing program in many strata now calls for it to recognize a few special numbers and know what to do in those cases, but to continue to play stupidly in other cases. However, by this time it is likely that the class has already discovered the full strategy. It may still be worthwhile to encourage at least some member to follow the original plan as an instructive joke. In this section we illustrate a general question-answer technique for classroom discussion to encourage habits of heuristic neatness in the students' own thinking.

A good exercise is to observe NIMPLAY in its present condition, and to collect and classify its mistakes. An example of a classification made by a student is:

- DUMB MISTAKES

- ★ There were five sticks and the machine took two.
(If the machine had any sense, it would leave the opponent with four.)
- ★ There were six or seven sticks and the machine did not leave four.

- SUPER DUMB MISTAKES

- ★ There were two or three sticks and the machine did not take all!

We shall write a procedure to avoid first "super dumb mistakes" and then "dumb mistakes."

- Question: What program form? Answer: TEST.
- Question: What do we test for? English answer: Whether there are one, two, or three sticks. Logo answer: TEST MEMBERP :STICKS [1 2 3].
- Question: What is the action if the test is passed? English answer: Take all the sticks. Logo answer: OUTPUT :STICKS.
- Question: What if it is not passed? English answer: Move just like before. Logo answer: MAKE "MOVE PICKRANDOM [1 2 3].

Now put this together to make a procedure to make the move:

- Question: What must the procedure know? Answer: :STICKS—so it needs an input.

```

TO MAKEMOVE :STICKS
TEST MEMBERP :STICKS [1 2 3]
IFTRUE [OUTPUT :STICKS]
IFFALSE [OUTPUT PICKRANDOM [1 2 3]]
END

```

The procedure is used in place of PICKRANDOM in GETMOVE. So don't forget to change GETMOVE!

Now extra lines can be added. For example:

```

TEST :STICKS = 5
IFTRUE [OUTPUT 1]

```

The smart player

By this time everyone should be very close to understanding the strategy, for example, in the following form:

- Question: How does the game end? Answer: When a player *leaves zero* sticks.

So let's try making the main actor be the number of sticks we leave. If we can leave zero that's great. But if we have more than three we can't. So we must think ahead.

- Question: What can we leave to help us leave zero next time? Answer: Four, because the opponent will leave one, two, or three.
- Question: What can we leave so as to be able to leave four next time? Answer: Eight.
- Question: So 0, 4, 8 are good numbers to shoot at for leaving. What others? Answer: 12, 16, . . .
- Question: How could you describe the numbers 0, 4, 8, 12, 16, . . . Answer: They are all divisible by 4.
REMAINDER :NUMBER 4 is 0.
- \$64 Question: If I give you :NUMBER, how can you use it to find the next number down divisible by 4? Answer:
Subtract REMAINDER :NUMBER 4.

So there we are! The smart, invincible Nim player is made by replacing MAKEMOVE by SMARTMOVE:

```

TO SMARTMOVE :STICKS
MAKE "REM REMAINDER :STICKS 4
IF :REM = 0 [OUTPUT 1]    It really doesn't matter in this case.
OUTPUT :REM
END

```

6.3.4. Frills and modifications

Write superprocedures or make additions to the present procedure to produce transcripts such as the one following.

NIM
 DO YOU KNOW HOW TO PLAY NIM?
 NO
 HERE ARE THE RULES: YOU WILL BE SHOWN A
 COLLECTION OF X'S.
 YOU MAY REMOVE 1, 2, or 3. THE PLAYER WHO TAKES
 THE LAST X
 WINS. THIS IS PROBABLY TOO VAGUE FOR YOU TO
 UNDERSTAND,
 BUT TRY PLAYING AND I'LL CORRECT YOUR MISTAKES.
 ARE YOU READY?
 I AM
 PLEASE SAY "YES" OR "NO"
 YES
 OK. NOW TELL ME THE NAME OF THE FIRST PLAYER.
 JOAN
 NOW THE NAME OF THE OTHER PLAYER
 COMPUTER
 HOW MANY STICKS DO YOU WANT TO START WITH?
 THIRTY-ONE
 I'M A DUMB COMPUTER. TYPE A PROPER NUMERAL.
 31
 JOAN TO PLAY.
 THERE ARE 31 STICKS.
 XX
 JOAN, TAKE 1, 2, OR 3
 3
 COMPUTER TO PLAY.
 THERE ARE 28 STICKS.
 XX
 I TAKE 1
 JOAN TO PLAY.
 THERE ARE 27 STICKS.
 XX
 TAKE 1, 2, OR 3
 3
 .
 .
 .

In addition to such frills, there are unlimited possibilities to play with the ideas in the procedure after it has been made to work. Here are three examples to illustrate the idea that the

project has not necessarily run out when the procedure is debugged:

- An interesting simple modification to the rules of the game is to change the 1-2-3 rule to a 1-2 rule or a 1-2-3-4-5 rule. Write a procedure that asks what rule is to be used, and then plays by that rule.
- Our stop rule was: the player who takes the last stick wins. Change this to: whoever takes the last stick loses. (The latter is the traditional form.)
- The game can be embedded in a more complex one, such as moving counters along marked paths on a board. If there is just one linear path, the problem is identical, but if branches are allowed, interesting complexities arise.

6.3.5. A listing of the NIMPLAY procedures

Here is a listing of the final form of the NIMPLAY procedures. Besides the three procedures listed below, the project also makes use of the READNUMBER procedure on page 100.

Type as one line. See page 9. →

Type as one line. See page 9. →

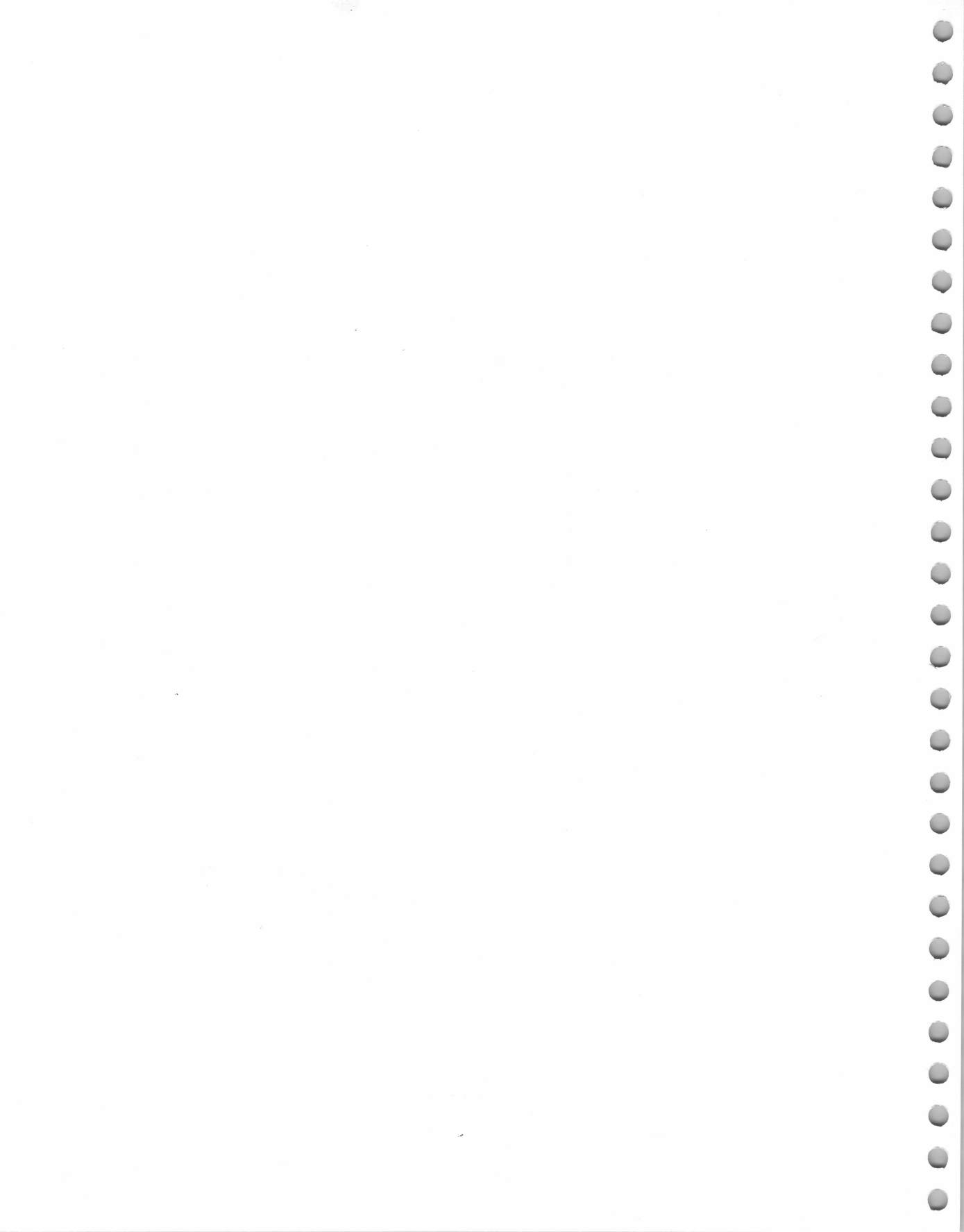
```

TO NIMPLAY :STICKS :PLAYER :OPPONENT
PRINT SENTENCE [THE NUMBER OF STICKS IS] :STICKS
PRINT SENTENCE :PLAYER
[TO PLAY. WHAT'S YOUR MOVE?]
MAKE "NEWSTICKS
:STICKS - (GETMOVE :PLAYER :STICKS)
TEST :NEWSTICKS = 0
IFTRUE [PRINT SENTENCE :PLAYER [IS THE WINNER!]]
IFTRUE [STOP]
NIMPLAY :NEWSTICKS :OPPONENT :PLAYER
END

TO GETMOVE :PLAYER :STICKS
TEST :PLAYER = [COMPUTER]
IFTRUE [MAKE "MOVE SMARTMOVE]
IFFALSE [PRINT [YOU MAY TAKE 1, 2, OR 3 STICKS]]
IFFALSE [MAKE "MOVE READNUMBER]
TEST MEMBERP :MOVE [1 2 3]
IFFALSE [OUTPUT GETMOVE :PLAYER :STICKS]
TEST :MOVE > :STICKS
IFTRUE [OUTPUT GETMOVE :PLAYER :STICKS]
OUTPUT :MOVE
END

TO SMARTMOVE
MAKE "REM REMAINDER :STICKS 4
IF :REM = 0 [OUTPUT 1]
OUTPUT :REM
END

```

CHAPTER 7

Writing Interactive Programs

We've already seen examples of Logo programs that use PRINT to print information on the display screen and programs that use READLIST to input information from the keyboard. This chapter reviews these commands and describes more elaborate ways of handling input and output. As an example, we show how to create "instant response" Logo systems for very young children. We also show how a Logo-based "dynaturtle" can be used to introduce elementary school children to computer projects involving motion and simple physics.

7.1. Controlling Screen Output

The Logo PRINT command, as used throughout the preceding chapters, is the main command for showing information on the display screen. PRINT takes a word or a list as an input, types it on the screen, and moves the cursor to the next line. (Remember that lists are printed without the outer brackets.) Although we have so far used PRINT with only one input, it can also take a variable number of inputs, providing that the word PRINT and the inputs are enclosed in parentheses, as explained in section 5.7.2. When PRINT is used with more than one input, it prints all the inputs on the same line, separated by spaces. For example:

```
MAKE "COLOR "GREEN
MAKE "TASTE "SOUR
(PRINT [THIS APPLE IS] :COLOR "AND :TASTE)
THIS APPLE IS GREEN AND SOUR
```

The command TYPE is just like PRINT, except that it does not move the cursor to a new line after printing. Compare

```
TO COUNTUP :X
PRINT :X
COUNTUP :X + 1
END

COUNTUP 1
```

```

1
2
3
.
.
.
TO COUNTUP1 :X
TYPE :X
TYPE ",
COUNTUP1 :X + 1
END

COUNTUP1 1
1,2,3, . . .

```

For more elaborate control of screen output, Logo provides the SETCURSOR command. This positions the cursor at a particular spot on the screen. SETCURSOR takes a list of two numbers as input, which indicates the column and row to which the cursor should be moved. Columns are numbered left to right, 0 through 39. Rows are numbered top to bottom, 0 through 23. After the cursor has been moved, subsequent PRINT commands continue printing from that position. The CURSOR command returns a list that specifies the current position of the cursor.

The CLEARTEXT command clears the text screen and moves the cursor to the top left of the text area.

7.2. Keyboard Input

The READLIST command is used to read input from the keyboard, as shown in section 5.4. READLIST causes the computer to wait for you to type in a line (terminated by RETURN) and then outputs that line as a list. Remember that what you type in will *always be interpreted as a list*. For example, if you type in a single word, READLIST returns a list containing that word:

```

MAKE "ANS READLIST
100
IF :ANS = 100 [PRINT "YES] [PRINT "NO]
NO
IF :ANS = [100] [PRINT "YES] [PRINT "NO]
YES

```

Using READLIST, you can implement a useful procedure that returns a word typed at the keyboard, obtained as the first element of the list returned by READLIST:

```

TO READWORD
OUTPUT FIRST READLIST
END

```

Compare the use of READLIST in the READNUMBER procedure on page 100.¹

In addition to the “line at a time” input from READLIST, Logo also provides “character at a time” input through the command READCHAR (abbreviated RC). READCHAR causes the computer to pause and wait for you to type in a single character (without RETURN) and then outputs the character that was typed:

```
MAKE "ANS READCHAR
X
IF :ANS = "X [PRINT "YES] [PRINT "NO]
YES
```

7.2.1. Example: instant response for very young children

The following program uses READCHAR to provide “instant response” control of the turtle for drawing:

```
TO INSTANT
COMMAND
INSTANT
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F [FORWARD 10]
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "C [CLEARSCREEN]
END
```

This program causes the turtle to move in response to individual keystrokes: F for forward, L for left, R for right, and C for clearing the screen and starting over. It can form a good tool for using computer graphics with very young children. This same instant-response mechanism is also useful in designing languages for use by the physically handicapped, for whom it is important to minimize the number of keystrokes required.

It is easy to increase the repertoire of this INSTANT language by adding additional lines to the COMMAND procedure. For example, if you want the S key to make the turtle draw a small square, you define a procedure called SQUARE (say, that draws a square of side 20) and add to COMMAND the line

```
IF :COM = "S [SQUARE]
```

Section 9.2.1 discusses more elaborate extensions to INSTANT.

¹ READWORD (and an abbreviation RW) is provided as part of the AIDS package which is normally loaded when Logo is started.

7.2.2. Keyboard control of an ongoing process

Notice that READLIST and READCHAR both make the computer *stop and wait* for something to be typed. Logo also allows you to write programs in which the keyboard is used to control an ongoing process. That is, if a character is typed at the keyboard, the program is able to respond to the character; but if nothing is typed, the program is able to keep running anyway. Such programs are implemented in Logo using the KEYP command. KEYP outputs TRUE or FALSE depending on whether a character has been typed at the keyboard. When KEYP is TRUE, the next READCHAR command returns the character that was typed; otherwise READCHAR has to wait until a character is typed.² For example, you can modify the INSTANT program on page 117 so that it makes the turtle move forward *continually*, turning right or left in response to the letters R and L typed at the keyboard. We'll call the resulting program DRIVE:

```
TO DRIVE
FORWARD 5
COMMAND
DRIVE
END

TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
END
```

The difference between this program and INSTANT is that the turtle goes forward every time, rather than only when an F is typed. Whereas the COMMAND program used by INSTANT calls READCHAR, the COMMAND program used in DRIVE calls READKEY. READKEY is a procedure that, if a character has been typed, outputs that character, and otherwise outputs the empty word. READKEY is implemented using KEYP:

```
TO READKEY
IF KEYP [OUTPUT READCHAR]
OUTPUT ""
END
```

² More specifically, characters typed at the keyboard are saved in an input buffer. READCHAR reads characters from the buffer one-by-one. KEYP outputs TRUE if the buffer is not empty. If Logo is doing a lot of processing in between characters, and if one types characters very quickly, a sequence of characters may build up in the buffer, and the program may seem to "fall behind" in its responses to the typed characters.

READKEY is a good example of a useful “primitive” that can be supplied by the teacher to students working on interactive programming projects.

7.3. Example: the Dynaturtle Program

DYNATURTLE is an extension of the Logo turtle, developed by Andy diSessa as a computer-based physics environment for elementary school students. It has also been used as an experimental setting for investigating the role of intuition in learning physics. See A. diSessa [3] for details. This section, based on a paper by Dan Watt, is a description of DYNATURTLE that can be used by students and teachers.

7.3.1. What is a dynamic turtle?

A *dynamic* turtle or *dynaturtle* behaves as though it were a rocket ship in outer space. To make it move you have to give it a *kick* by “firing a rocket.” It then *keeps moving* in the *same* direction until you give it another kick. When you change its direction, it does not move in the new direction until you give it a new kick. Its new motion is a combination of the old motion and the motion caused by the new kick. You may need to experiment with dynamic commands for a while before you understand how the dynaturtle works.

To use the dynaturtle, you will need the procedures in this section. Here is the main procedure:

```
TO DT
MOVETURTLE
COMMAND
DT
END
```

The procedure DT moves the turtle (if you have given it a kick), checks to see if you’ve typed a command, and then starts doing DT all over again. It keeps running until the procedure is stopped.

Here are four other procedures you will need:³

```
TO MOVETURTLE
SETPOS SE ( XCOR + :VX ) ( YCOR + :VY )
END
```

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "K [KICK]
END
```

³ The KICK procedure uses the trigonometric functions SIN and COS in order to change the turtle’s velocity, which can be thought of as a vector (VX , VY). This procedure would normally be used by elementary school children as a black box.

```

TO READKEY
IF KEYP [OUTPUT READCHAR]
OUTPUT "
END

TO KICK
MAKE "VX :VX + SIN HEADING
MAKE "VY :VY + COS HEADING
END

```

To start the dynaturtle, you need a procedure to initialize the dynaturtle's position and velocity:

```

TO STARTUP
CLEARSCREEN
MAKE "VX 0
MAKE "VY 0
END

```

To try out the dynaturtle, type

```

STARTUP
DT

```

At first the turtle will stay at the center of the screen. The COMMAND procedure allows three different commands at present. Later you can change them in any way you like.

- If you type R the turtle will turn right 30 degrees.
- If you type L the turtle will turn left 30 degrees.
- If you type K you will give the turtle a *kick* in the direction it is heading.

The turtle will now keep moving in the direction it started until you give it another kick in some direction.

7.3.2. Activities with a dynaturtle

Start the dynaturtle moving by typing

```

STARTUP
DT

```

and then typing the K key for "kick."

- Make the dynaturtle move in a different direction by typing the R or L key.
- Make the dynaturtle move horizontally across the screen.
- Make the dynaturtle go faster.

- Make the dynaturtle go slower, without changing direction.
- Before you start the dynaturtle, draw a marker (a small box or "x") somewhere on the screen. Then start the dynaturtle and see if you can move the turtle to the marker. If the marker is easy for the dynaturtle to get to, move it over a little and try again.
- Start the dynaturtle from the center of the screen. Can you make it stop?
- Draw a circular "racetrack" on the screen and see if you can "drive" the dynaturtle around the track.
- Move the dynaturtle to the marker, and make it stop there.

When you try these activities you may find that some of them are harder than you thought. The problems you have making the dynaturtle do what you want it to do are similar to the problems astronauts would have moving around in outer space, or maneuvering a rocket to connect up with a space platform or land on the moon.

7.3.3. Changing the dynaturtle's behavior

After some experimentation with the dynaturtle, you may want to make changes in the dynaturtle procedures. Since changes in the dynaturtle's behavior are controlled by the COMMAND procedure, you can start by changing that procedure:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "K [KICK]
END
```

If you like, you can change the *angle* the dynaturtle rotates when you type R or L by changing the 30 in COMMAND to another number. You could add commands to raise or lower the dynaturtle's pen:

```
IF :COM = "U [PENUP]
IF :COM = "D [PENDOWN]
```

Your COMMAND procedure would now look like this:


```

TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "K [KICK]
IF :COM = "U [PENUP]
IF :COM = "D [PENDOWN]
END

```

Of course you can change the key names for carrying out the commands by changing the letters on the keyboard. Some people like to have the right and left keys next to each other on the keyboard. If you choose A for "left" and S for "right," then the COMMAND procedure becomes:

```

TO COMMAND
MAKE "COM READKEY
IF :COM = "S [RIGHT 30]
IF :COM = "A [LEFT 30]
IF :COM = "K [KICK]
IF :COM = "U [PENUP]
IF :COM = "D [PENDOWN]
END

```

Another change could be to make the *force* of the kick a variable. If you did this you would have to change the KICK procedure and the STARTUP procedure as well as COMMAND.

```

TO KICK :FORCE
MAKE "VX :VX + :FORCE * ( SIN HEADING )
MAKE "VY :VY + :FORCE * ( COS HEADING )
END

```

You would also have to add a line to STARTUP to set the starting value for the force:

```

TO STARTUP
MAKE "VX 0
MAKE "VY 0
MAKE "FORCE 1
END

```

You can choose any value you want for the starting force.

You would now have to change the KICK line in the COMMAND procedure to read

```

IF :COM = "K [KICK :FORCE]

```

Also, you could add two more commands (say, F and S) that increased and decreased the force. The COMMAND procedure would now be

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RT 30]
IF :COM = "L [LT 30]
IF :COM = "K [KICK :FORCE]
IF :COM = "U [PENUP]
IF :COM = "D [PENDOWN]
IF :COM = "F [MAKE "FORCE :FORCE + 1]
IF :COM = "S [MAKE "FORCE :FORCE - 1]
END
```

Now try out the dynaturtle with some of these changes, and see what can happen.

Some other possible changes:

- Add a "reverse kick" command that makes the dynaturtle move more slowly.
- Add commands that make the turtle *print* its speed, heading, and kick force.

7.4. Input from the Paddles

Besides using the keyboard for input in interactive programs, Logo can also make use of the paddles that can be connected to the Apple computer. One can connect up to four paddles, which in Logo are numbered 0 through 3. The PADDLE command takes a number 0 through 3 as input. It outputs a number between 0 and 255, which varies as the dial on the paddle is turned.

As a simple example of how to use PADDLE, you can write a "sketching" program that uses paddles 0 and 1 to control the horizontal and vertical motion of the turtle.

```
TO SKETCH
SETPOS SENTENCE PADDLE 0 PADDLE 1
SKETCH
END
```

One problem with this program is that the turtle runs off the screen when the paddle dials are turned too high. Also, the turtle's *x* and *y* coordinates can never be negative. To improve the program, you can first subtract the paddle input from 128 to produce a number between -128 and 128, and then multiply by appropriate scale factors to transform the -128 to 128 range into a good range for screen coordinates, say, -150 to 150 for *x* and -90 to 90 for *y*.

Type as one line. See page 9.→

```
TO SKETCH1
SETPOS SE ((128 - PADDLE 0) * 150/128)
          ((128 - PADDLE 1) * 90/128)
SKETCH1
END
```

The paddles supplied with the Apple have buttons as well as dials. (One can connect four paddles to an Apple, but only the buttons on the first three of them can be sensed by the computer.) The Logo command `BUTTONP` takes a number 0 through 2 as input, and outputs `TRUE` or `FALSE` depending on whether the button on the corresponding paddle is pressed.

As an example, you can extend the `SKETCH1` program so that pressing the button on paddle 0 causes the turtle to begin drawing in a different color, and pressing the button on paddle 1 clears the screen.

Type as one line. See page 9.→

```
TO SKETCH2
IF (BUTTONP 0) [CHANGECOLOR]
IF (BUTTONP 1) [CLEARSCREEN]
SETPOS SE ((128 - PADDLE 0) * 150/128)
          ((128 - PADDLE 1) * 90/128)
SKETCH2
END

TO CHANGECOLOR
SETPC (RANDOM 6)
END
```

CHAPTER 8

Inputs, Outputs, and Recursion

One important difference between Logo and other common programming languages is that, in Logo, words and lists can be used as inputs and outputs to procedures. Therefore, when you program in Logo, you can work in terms of operations that act on entire words and lists, rather than only on individual numbers and characters. Consider the `DOUBLE.LIST` procedure that was introduced on page 85:

```
TO DOUBLE.LIST :X
  OUTPUT SENTENCE :X :X
END

PRINT DOUBLE.LIST [DO RE MI]
DO RE MI DO RE MI
```

The importance of making this procedure `OUTPUT` its result is not merely so that you can `PRINT` the result, but so that you can use the result as an input to another procedure that can perform further operations. For instance, if you have a procedure `REVERSE.LIST` that reverses a list (as we shall write in section 8.1 below) then you can produce the reverse of the double of a list `X` by

```
REVERSE.LIST DOUBLE.LIST :X
```

More generally, you can construct complex operations on words and lists as successions of procedures, each of which performs a simple operation and passes the result on to the next procedure. To obtain an operation that removes the last word from a list, reverses what is left, and doubles the result, you can write:

```
DOUBLE.LIST (REVERSE.LIST (BUTLAST :X))
```

as in the command

```
PRINT DOUBLE.LIST (REVERSE.LIST (BUTLAST [A B C D]))
C B A C B A
```

which produces the chain of operations shown in figure 8.1. Building up complex operations by combining simpler operations is common programming practice in working with numbers. For example, it is natural to think of computing $(x-1)^2 + 1$ in terms of the simpler operations of subtracting 1 from a number, squaring the result, and adding 1. Logo enables you to use the same kind of strategy in dealing with words and lists.

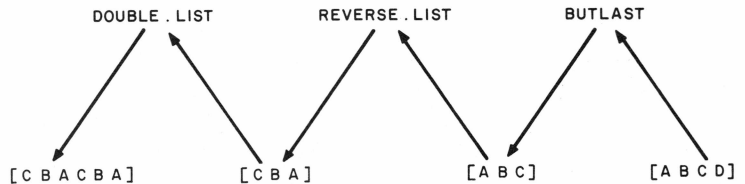


Figure 8.1: Chain of inputs and outputs in a sequence of list operations.

The ability to construct complex operations as combinations of simpler ones is particularly powerful when combined with another problem solving strategy: one can often solve a problem by first solving a simpler problem of the same sort and then making a simple modification to the answer. For example, suppose you want to write a procedure that counts the number of words in a list. Imagine that you already know how many words are in the BUTFIRST of the list. Then you could solve your original problem by simply taking the number of words in the BUTFIRST and adding 1. We will see in section 8.2.1 below that this method leads to a simple recursive procedure that counts the number of words in a list.

Recursive procedures, in general, are the computational analogues of strategies that attack problems by reducing them to simpler problems of the same sort. Given the ability of Logo procedures to manipulate words and lists, this implies that many useful word and list operations can be implemented as surprisingly simple recursive procedures. This chapter examines a few of them. We consider first a number of procedures involved with reversing words and lists and checking for palindromes. Then we discuss operations that select words from lists and test whether a word is a member of a list. Finally, we show how the problem of converting numbers from one base to another can be solved by a simple recursive strategy.

8.1. Reversing Words and Lists

This section shows how to use recursion to write a procedure `REVERSE` that takes a word as input and outputs the word with the characters reversed. That is to say, `REVERSE` behaves as follows:

```
PRINT REVERSE "STRESSED
DESSERTS
```

```
PRINT REVERSE "RUMPLESTILTSKIN
NIKSTLITSELPUR
```

```
PRINT REVERSE REVERSE "RUMPLESTILTSKIN
RUMPLESTILTSKIN
```

Logo's `LAST` and `BUTLAST` operations, which encourage thinking about a word in terms of its last character and the rest of the word, suggest a recursive strategy for implementing the `REVERSE` procedure. It is based on the following idea: suppose you are given a word, say `BIRD`, and you are asked to reverse it. Now imagine that you have somehow managed to generate the reverse of all but the last character of the word—`RIB`. Then all you have to do to reverse the original word is to take the last character, `D`, and place it at the front of what you already have—`DRIB`. This reduces the problem of reversing a word to the problem of reversing a shorter word, namely, the `BUTLAST` of the word. That problem reduces in turn to reversing a still shorter word, namely, the `BUTLAST` of the `BUTLAST`, and so on, with shorter and shorter words. You can diagram this process as follows:

`REVERSE "BIRD` is $D \leftrightarrow \text{RIB}$

or the last character of `BIRD` added in front of `RIB`. But

`RIB` is `REVERSE "BIR` which is $R \leftrightarrow \text{IB}$

or the last character of `BIR` added in front of `IB`. But

`IB` is `REVERSE "BI` which is $I \leftrightarrow \text{B}$

or the last character of `BI` added in front of `B`. Now put all these together:

```
REVERSE "BIRD = D ↔ (REVERSE "BIR )
                = D ↔ R ↔ (REVERSE "BI )
                = D ↔ R ↔ I ↔ (REVERSE "B )
                = D ↔ R ↔ I ↔ B
```

This strategy, reducing the problem of reversing a word to the

problem of reversing BUTLAST of the word, leads to the following recursive procedure:

```
TO REVERSE :X
  OUTPUT WORD (LAST :X) (REVERSE BUTLAST :X)
END
```

However, if you execute this procedure, it will not work. Instead, it produces the error

LAST DOESN'T LIKE AS INPUT

The problem is that there is no *stop rule*. Nothing tells REVERSE to stop taking LASTs and BUTLASTs of its input, and Logo finally complains when the procedure attempts to take LAST of the empty word. At some point, REVERSE should simply output an answer directly without reducing the problem to one of reversing a still shorter word. For example, if the word to be reversed is a single character, then REVERSE of the word is the word itself, so you can add to REVERSE the stop rule:

```
IF BUTLAST :X = " [OUTPUT :X]
```

where BUTLAST :X being empty signals that :X consists of a single character. So here is the complete procedure:

```
TO REVERSE :X
  IF BUTLAST :X = " [OUTPUT :X]
  OUTPUT WORD (LAST :X) (REVERSE BUTLAST :X)
END
```

An alternative stop rule allows the recursion to proceed one step further to the point where the input is empty, and then to return the empty word:

```
TO REVERSE :X
  IF :X = " [OUTPUT ""]
  OUTPUT WORD (LAST :X) (REVERSE BUTLAST :X)
END
```

This has the same effect as the previous version, as can be seen by observing what happens when this procedure is used to reverse a one-character word:

```
REVERSE "B = B ↔ REVERSE "
            = B ↔ {empty}
            = B
```

Similar reasoning can be applied to produce a procedure REVERSE.LIST which takes a list as input and returns the list of words in reverse order, as in

```
PRINT REVERSE.LIST [OH SAY CAN YOU SEE]
SEE YOU CAN SAY OH
```

As before, the problem reduces to combining the LAST of the input list with REVERSE.LIST of the BUTLAST, only, since you will be combining lists rather than words, you should use SENTENCE rather than WORD to form the combination. The stop rule, like the one above, checks for the list being reduced to empty, only this time you must check for the empty list rather than the empty word. Here is the procedure:

```
TO REVERSE.LIST :X
IF :X = [ ] [OUTPUT [ ]]
OUTPUT SENTENCE (LAST :X) (REVERSE.LIST BUTLAST :X)
END
```

You can combine REVERSE and REVERSE.LIST to obtain a procedure REVERSE.ALL that takes a list as input and returns a list of the words in reverse order, with each word reversed, as well:

```
PRINT REVERSE.ALL [OH SAY CAN YOU SEE]
EES UOY NAC YAS HO
```

All you need to do to implement REVERSE.ALL is to modify REVERSE.LIST so that it REVERSEs the LAST word of its input before combining that with the result of reversing the BUTLAST:

```
TO REVERSE.ALL :X
IF :X = [ ] [OUTPUT [ ]]
OUTPUT SENTENCE (REVERSE LAST :X)
                  (REVERSE.ALL BUTLAST :X)
END
```

Type as one line. See page 9.→

The reasoning that led to these procedures is typical of most recursive procedures that involve words and lists:

- There is a *reduction step* that reduces the problem to a similar problem on a shorter word or list (usually the BUTFIRST or BUTLAST of the input).
- There is a *stop rule* that checks for some simple case (usually the input being reduced to a single element, or

to the empty word or the empty list).¹

Note the use of recursion and the fact that each procedure must explicitly OUTPUT its result to the procedure that calls it, as noted on page 80. Figure 8.2 shows the pattern of inputs and outputs that results from executing

REVERSE "BIRD

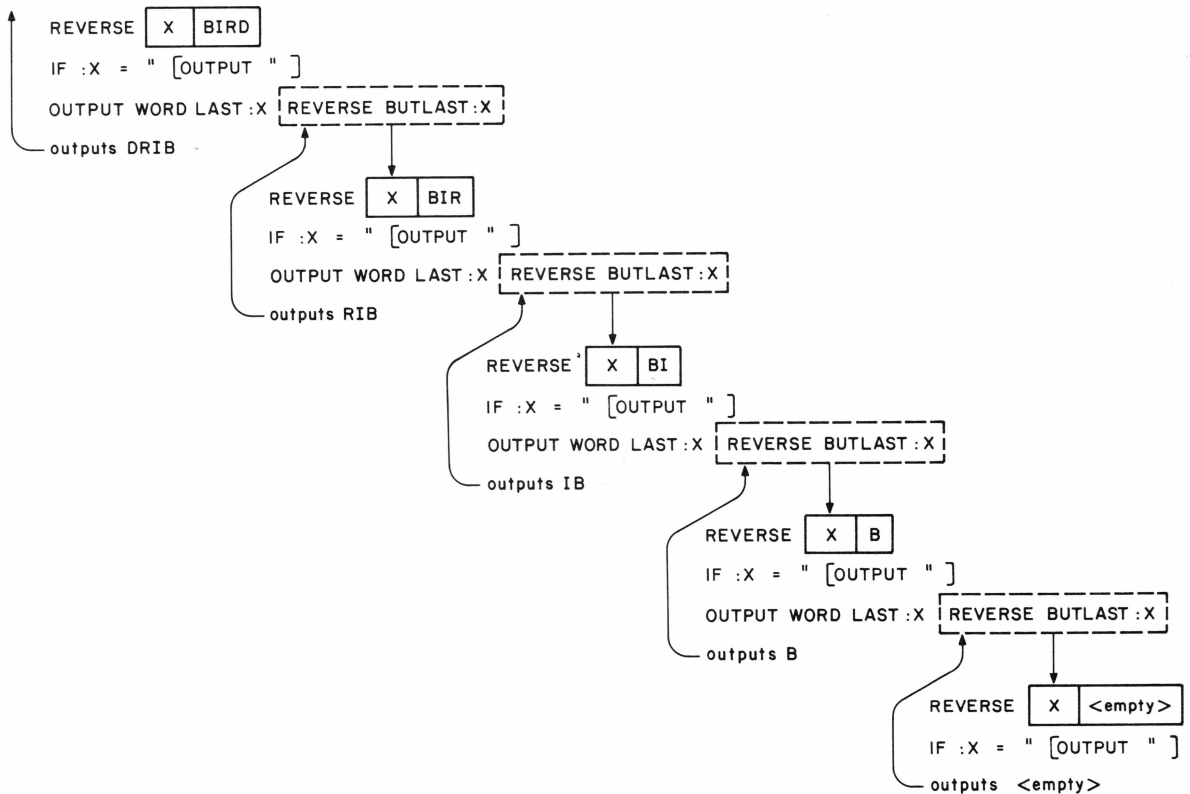


Figure 8.2: Procedure calls in executing REVERSE "BIRD.

¹ Notice that in the actual procedure, the stop rule is written before the reduction step. But when you formulate a recursive solution, you most likely discover the reduction step first and then design an appropriate stop rule.

Finding Palindromes

You can apply REVERSE to the problem of checking whether a word is a palindrome, that is, whether the word reads the same backwards as forwards. For example:

```
PRINT PALINDROME? "ABA
TRUE
PRINT PALINDROME? "ABC
FALSE
PRINT PALINDROME? "RACECAR
TRUE
```

Since a word is a palindrome precisely when it is equal to its reverse, you can easily write a Logo predicate that tests for palindromes:²

```
TO PALINDROME? :X
IF :X = REVERSE :X [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

One interesting project that involves palindromes has to do with numbers: Start with any integer. If it is a palindrome, then stop. Otherwise add the number to its reverse. If that is a palindrome, then stop, and so on. The question is, if you start with any number, will the process always eventually produce a palindrome?

You can write a Logo program to carry out this process:

```
TO MAKE.PALINDROME :NUMBER
PRINT :NUMBER
IF PALINDROME? :NUMBER [STOP]
MAKE.PALINDROME :NUMBER + REVERSE :NUMBER
END
```

```
MAKE.PALINDROME 78
78
165
726
1353
4884
```

² Notice that since the Logo predicate = itself outputs either TRUE or FALSE, you can write this same procedure more simply as

```
TO PALINDROME? :X
OUTPUT (:X = REVERSE :X)
END
```

```
MAKE.PALINDROME 16793
16793
56554
102119
1013320
1246421
```

This is a good program to experiment with. As a project, search for numbers less than 1000 that do not seem to produce palindromes in a very few steps.

8.2. Recursive Procedures that Manipulate Lists

Logo's list operations FIRST, LAST, BUTFIRST, and BUTLAST are the basic ways to reduce lists to simpler lists. List operations can often be implemented by means of recursive strategies that reduce the problem of performing some operation on a list to the problem of performing a similar operation on the BUTFIRST (or BUTLAST) of the list. This section considers three useful list operations that are included as primitives in Apple Logo:

- COUNT, which takes a list as input and returns the number of items in the list
- ITEM, which takes a number n and a list as input and returns the n th item of the list
- MEMBERP, which takes an item and a list as input and returns TRUE or FALSE, depending on whether the item is a member of the list

and shows how these can be implemented in Logo as recursive procedures using FIRST and BUTFIRST. In order to avoid confusion with the primitive operations, we'll call our three corresponding procedures LENGTH, PICK, and MEMBER?.

8.2.1. The LENGTH procedure

The LENGTH procedure takes a list as input and outputs the number of items in the list:

```
PRINT LENGTH [HAVE A NICE DAY]
4
PRINT LENGTH [HAVE]
1
PRINT LENGTH [ ]
0
```

The implementation of LENGTH uses a simple recursive strategy:

- The length of a list is 1 plus the length of the BUTFIRST of the list.
- The length of the empty list is 0.

That is to say, you can compute the length of a list by first computing the length of the BUTFIRST of the list and adding 1 to the result. But computing the length of the BUTFIRST reduces to computing the length of a still shorter list, and so on, until you are reduced to computing the length of the empty list, which is 0. Here is an example of this process in action:

```

LENGTH [DO RE MI FA]
= 1 + (LENGTH [RE MI FA])
= 1 + (1 + (LENGTH [MI FA]))
= 1 + (1 + (1 + (LENGTH [FA])))
= 1 + (1 + (1 + (1 + (LENGTH [ ]))))
= 1 + (1 + (1 + (1 + 0)))
= 1 + (1 + (1 + 1))
= 1 + (1 + 2)
= 1 + 3
= 4

```

Although the process seems complicated when it is “unwound” as in the example above, you can readily express it as a recursive rule:

- *Reduction Step:* Output 1 plus the length of the BUTFIRST of the list.
- *Stop Rule:* If the list is empty output 0.

This leads in turn to a recursive procedure:

```

TO LENGTH :X
IF :X = [] [OUTPUT 0]
OUTPUT 1 + LENGTH BUTFIRST :X
END

```

Figure 8.3 shows the pattern of procedure calls that results from using this procedure to compute

```

LENGTH [DO RE MI FA]

```

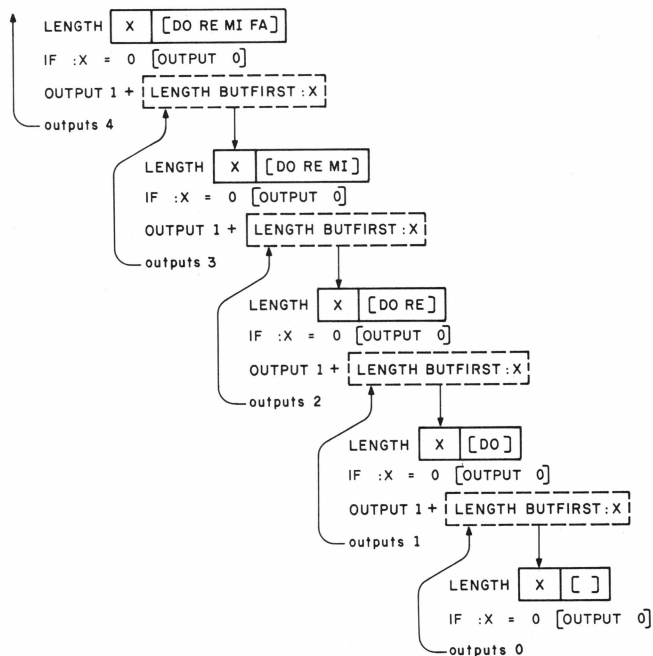


Figure 8.3: Procedure calls in executing LENGTH [DO RE MI FA].

8.2.2. The PICK procedure

One of the most useful operations to have in working with lists is the ability to select an item from a list. Consider the problem of writing a procedure PICK that takes a number and a list as inputs and outputs the designated item from the list: if the number is 1, PICK outputs the first item in the list; if the number is 2, PICK outputs the second item in the list; and so on.

There is a recursive strategy for computing PICK in terms of the operations FIRST and BUTFIRST. Just as with LENGTH, you can reduce the problem of picking an item from a list to the problem of picking an item from the BUTFIRST of the list: the n th item of a list is the same as the $(n - 1)$ st item of the BUTFIRST of the list. The recursive plan is:

- *Reduction Step:* To PICK the n th item from a list, then PICK the $(n - 1)$ st item of the BUTFIRST of the list.
- *Stop Rule:* If $n = 1$ then output the FIRST item in the list.

This strategy can be expressed as the following Logo procedure:

```

TO PICK :N :X
IF :N = 1 [OUTPUT FIRST :X]
OUTPUT PICK (:N - 1) (BUTFIRST :X)
END

```

Figure 8.4 shows the chain of procedure calls and the inputs and outputs in executing:

PICK 3 [A B C D]

where picking the third item of [A B C D] reduces to picking the second item of [B C D] which reduces to picking the first item of [C D], which is C.³

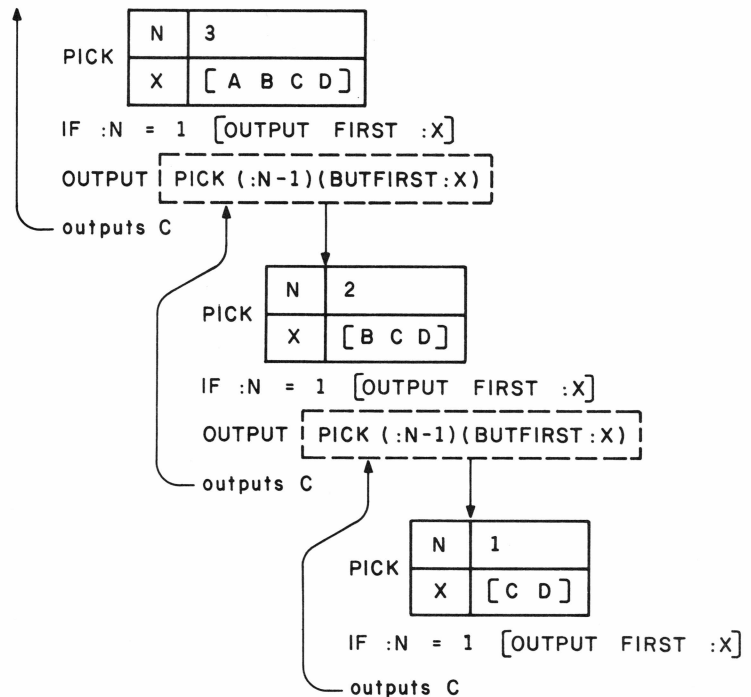


Figure 8.4: Procedure calls in executing PICK 3 [A B C D].

8.2.3. The MEMBER? predicate

The final operation we shall discuss is the `MEMBER?` predicate, which takes a word and a list as inputs and checks whether the word is a member of the list, outputting `TRUE` or `FALSE` accordingly. The recursive strategy here is that it is easy to check if the desired word is the `FIRST` item in the list. If it is, then `MEMBER?` should output `TRUE`. If not, you check to

³ If you call `PICK` with `N` larger than the length of the list, then the procedure signals an error. For example, trying to pick the 5th item of [A B C D] reduces to the 4th item of [B C D], the 3rd item of [C D], the 2nd item of [D], and finally `PICK` is called with `N` equal to 1 and `X` equal to the empty list. At this point, `PICK` tries to compute `FIRST` of `X`. This signals an error, since `FIRST` does not operate on the empty list.

see if the word is in the BUTFIRST of the list, and so on. If the list ever becomes empty, you have run out of elements to check the word against, so MEMBER? should output FALSE. The resulting procedure is

```
TO MEMBER? :WORD :LIST
IF :LIST = [] [OUTPUT "FALSE]
IF :WORD = (FIRST :LIST) [OUTPUT "TRUE]
OUTPUT MEMBER? :WORD (BUTFIRST :LIST)
END
```

Converting to Pig Latin

As an example using MEMBER? and recursion, you can write a program that converts a sentence to Pig Latin. For each word in the sentence, you must move the leading consonants to the end of the word and add on "ay" as in

Isthay entencesay isay inay igpay atinlay.

Since you need to strip off consonants, it is useful to have a predicate that checks whether a word begins with a vowel. That's easily done:

```
TO BEGINS.WITH.VOWEL? :W
OUTPUT MEMBER? (FIRST :W) [A E I O U]
END
```

Notice that this outputs TRUE or FALSE because MEMBER? outputs TRUE or FALSE.

Here's a program that converts a single word to pig Latin:

```
TO PIG :W
TEST BEGINS.WITH.VOWEL? :W
IFTRUE [OUTPUT WORD :W "AY]
IFFALSE [OUTPUT PIG WORD (BUTFIRST :W) (FIRST :W)]
END
```

The cleverness in PIG is the recursive call that ensures that PIG will keep stripping letters off the front of the word until it reaches a vowel. To better understand this point, you should draw a diagram that gives the sequence of recursive calls in computing

```
PRINT PIG "STRING
INGSTRAY
```

Now, if you work word-by-word, you can convert an entire sentence. The trick is to think recursively again:

```
TO PIGL :S
IF :S = [] [OUTPUT []]
OUTPUT SENTENCE (PIG FIRST :S) (PIGL BUTFIRST :S)
END

PRINT PIGL [THIS IS ANOTHER RECURSIVE PROCEDURE]
          ISTHAY ISAY ANOTHERAY ECURSIVERAY OCEDUREPRAY
```

The strategy used in PIGL is a standard way to “do something to every item in a list.” The idea is to reason as follows. Suppose you have already converted the words in the BUTFIRST of the list. Then you need only SENTENCE this with the result of converting the first word in the list, and you are done. In this way, the problem of converting the entire list reduces to converting BUTFIRST of the list, which reduces to BUTFIRST of *that* list, and so on, and so on. Finally the problem is reduced to that of converting the empty list, for which the answer is empty.

8.3. Radix Conversion

As a final example of a problem that seems difficult but has a simple recursive solution, we consider the problem of converting an integer written in base 10 notation to some other base, say, base 8. For instance, we would like to find that 65 base ten is written as 101 in base 8, 100 base ten is 144 base 8, 1000 base 10 is 1750 base 8, and so on.

There is a clever recursive strategy for solving this problem. Suppose that n is some integer and that you want to find the string of digits that represents n in base 8. Think about what such a representation means. For example, to say that 100 base ten is written as 144 base 8 means that

$$100 = 1 \times 8 \times 8 + 4 \times 8 + 4 = 144 \text{ base } 8$$

The key insight is that it is easy to find the *last* digit of the string of digits that represents n : This is just the remainder when n is divided by 8:

REMAINDER 100 8 is 4

Now suppose you take the string of digits that represents n and strip off the last digit. In terms of base 8 representation, that corresponds to shifting everything one place to the right and dropping the last digit. But this corresponds precisely to dividing the number by 8 and dropping the remainder. That is to say, if you take the string of digits that represents n in base 8 and leave off the last digit, what you are left with is the

string of digits that represents the integer quotient of n by 8, written in base 8:

QUOTIENT 100 8 is 12, and 12 represented in base 8 is 14, which is all but the last digit of 144.

So now you have a simple description of the string of digits that represents n in base 8

- The LAST digit is the remainder of n by 8.
- The BUTLAST of the string is the base 8 representation of integer quotient of n by 8.

So the problem of representing n in base 8 reduces to representing the quotient of n by 8 in base 8, which reduces further, and so on. The reductions stop when you reach a quotient of 0, which is represented as 0.

So here is how to generate n in base 8:

- If $n = 0$ the result is 0.
- Otherwise
 - ★ Find the digits that represent the quotient of n by 8, and
 - ★ append to these the remainder of n divided by 8.

Using the Logo WORD operation to glue numbers together, this strategy translates into the procedure:

```
TO BASE8 :N
IF :N = 0 [OUTPUT 0]
OUTPUT WORD (BASE8 (QUOTIENT :N 8))
(REMAINDER :N 8)
END
```

Type as one line. See page 9. →

The only slight bug in this procedure is that it generates an extra zero to the left of each non-zero number. You can fix this by providing another procedure to remove the extra zero:

```
TO CONVERT.TO.BASE.8 :N
IF :N = 0 [OUTPUT 0]
OUTPUT BUTFIRST BASE8 :N
END
```

Of course, there is nothing special about base 8. You can convert to any base less than 10 in the same way:

```
TO CONVERT :N :B
IF :N = 0 [OUTPUT 0]
OUTPUT BUTFIRST (BASE :N :B)
END

TO BASE :N :B
IF :N = 0 [OUTPUT 0]
OUTPUT WORD (BASE (QUOTIENT :N :B) :B)
          (REMAINDER :N :B)
END
```

Type as one line. See page 9. →

For example

```
PRINT CONVERT 1000 2
1111101000
```

For bases larger than 10, you can use the same strategy, except that you need “digits” to represent the single-digit numbers larger than 10. For instance, in base 16 you can represent 10 by the letter A and 11 by the letter B, and so on.⁴ So you can represent single digit numbers in base 16 by using the following procedure:

```
TO DIG :N
IF :N < 10 [OUTPUT :N]
IF :N = 10 [OUTPUT "A"]
IF :N = 11 [OUTPUT "B"]
IF :N = 12 [OUTPUT "C"]
IF :N = 13 [OUTPUT "D"]
IF :N = 14 [OUTPUT "E"]
IF :N = 15 [OUTPUT "F"]
END
```

Then a procedure for printing numbers in bases up to 16 could use:

```
TO CONVERT :N :B
IF :B > 16 [PRINT [ERROR: BASE TOO LARGE]]
IF :N = 0 [OUTPUT 0]
OUTPUT BUTFIRST BASE :N :B
END
```

⁴ This is the “hexadecimal” notation commonly used for specifying computer memory addresses.

Type as one line. See page 9. →

```
TO BASE :N :B
IF :N = 0 [OUTPUT 0]
OUTPUT WORD (BASE (QUOTIENT :N :B) :B)
          (DIG (REMAINDER :N :B))
END

PRINT CONVERT 20000 16
4E20

PRINT CONVERT 20000 12
B6A8
```

CHAPTER 9

Advanced use of lists

We've seen how words can be grouped together into Logo lists. But lists in Logo can be used for more than just collecting words. For example, the random sentence generator of section 6.2 picked its nouns from a list:

```
MAKE "NOUNS [DOGS CATS CHILDREN TIGERS]
```

Suppose, however, you want to make sentences using "nouns" that aren't single words. For example, you may want to make sentences about dogs, cats, children, tigers, and pack rats. You can't do this by adding the two words PACK RATS to the above list as in

Type as one line. See page 9. →

```
MAKE "NOUNS
      [DOGS CATS CHILDREN TIGERS PACK RATS]
```

because making sentence whose nouns are words picked at random from this list of 6 items would give results including things like

```
PACK LAUGH
RATS RUN
```

```
.
.
.
```

What you need to do is to take the two words PACK RATS and group these together as a *single* item within the list of nouns. You can do this in Logo by

Type as one line. See page 9. →

```
MAKE "NOUNS
      [DOGS CATS CHILDREN TIGERS [PACK RATS]]
```

What you have now is a list of five items. The first four items in the list are words: DOGS, CATS, CHILDREN, TIGERS. The fifth item in the list is *itself* a list [PACK RATS] consisting of two words, PACK and RATS. When you pick items from the list NOUNS you may get a single word like DOGS, or you may

get the two-word list [PACK RATS]. This new value of NOUNS gives the desired results in the sentence-generating program of section 6.2:

DOGS BITE
PACK RATS LAUGH

.
.
.

The general point here is that in Logo *the items in a list can be not only words, but also other lists*.

9.1. Hierarchical Structures

If you think of a list of words as a simple list (or one-level list), then the NOUNS list above can be considered to be a two-level list, that is, a list with an element that is itself a list. But there is no reason to stop there. In general, you can have lists whose items are themselves lists whose items are lists, and so on. And this general notion of a list in Logo provides lots of power and flexibility in dealing with complex structures. For example, figure 9.1 shows a *tree structure* that represents part of the organization of the U.S. government.

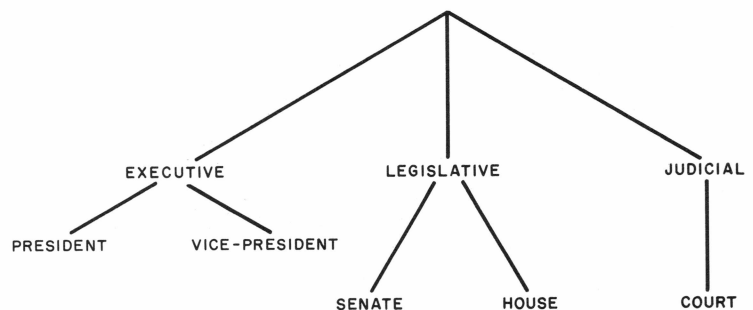


Figure 9.1: Hierarchical organization of U.S. Government.

From our point of view, the important thing about this structure is that it is a *hierarchy*, that is, it consists of parts that themselves consist of parts, and so on. You can represent the tree structure in figure 9.1 as the Logo list

```
[ [EXECUTIVE [PRESIDENT VICE-PRESIDENT]]
  [LEGISLATIVE [SENATE HOUSE]]
  [JUDICIAL [COURT]] ]
```

This is a list of three items.¹ The first item, which is the list

[EXECUTIVE [PRESIDENT VICE-PRESIDENT]]

is itself a list of two items, of which the first is the word EXECUTIVE and the second is a list of two words, and so on.

Logo's use of lists is adapted from the programming language Lisp, which was developed for research in artificial intelligence. Lists have proved to be indispensable in programs that deal with symbol manipulation and complex data structures, and their presence in Logo and Lisp is largely responsible for the fact that programming in these languages is very different from working in languages like BASIC and Fortran. In those languages, complex data structures must be encoded in terms of numbers, character strings, and arrays. Lists, however, allow many kinds of complex hierarchical structures to be represented directly, and therefore lists play a major role in computer applications dealing with complex data structures. In particular, they are the workhorse of most programs that are heavily involved with symbolic expressions, rather than just numerical data. The projects in section 9.3 illustrate how lists are used in this way. However, this hardly scratches the surface of what can be done. The book by Winston and Horn [16] provides many examples of the uses of lists in symbol manipulation, in the context of the language Lisp.

9.1.1. List operations

We've already seen how to use the Logo operations FIRST, LAST, BUTFIRST, BUTLAST, and SENTENCE for working with "simple" lists of words. These same operations extend to work with complex lists, as well. For example, suppose you create a complicated list:

MAKE "TRY [[A B C] D [E F]]

TRY is a list of three items, the list [A B C], the word D, and the list [E F].

PRINT :TRY
[A B C] D [E F]

Note how TRY is printed. Logo always prints lists without the outermost pair of brackets.

The operations FIRST and LAST output, as usual, the first and last items in a list. In a complex list these items may themselves be lists:

¹ Note how the list is printed, lining up its three elements in order to make its structure more readable.

```
PRINT FIRST :TRY
A B C
PRINT LAST :TRY
E F
```

BUTFIRST outputs the list consisting of all elements but the first, and BUTLAST outputs the list consisting of all elements but the last:

```
PRINT BUTFIRST :TRY
D [E F]
PRINT BUTLAST :TRY
[A B C] D
```

Keep in mind that the operations output, in general, new lists, to which you can apply further operations. For example,

```
FIRST FIRST :TRY
```

is the first item of the first item of TRY, which is the first item of [A B C], which is A.

```
FIRST LAST :TRY
```

is the first item of the last item of TRY, which is the first item of [E F], which is E.

```
FIRST BUTFIRST :TRY
```

is the first item of the BUTFIRST of TRY, which is the first item of [D [E F]], which is D. (In general, FIRST of BUTFIRST of any list is the second item of the list.)

```
FIRST BUTFIRST LAST :TRY
```

is the FIRST of the BUTFIRST of the LAST of TRY, which is the FIRST of the BUTFIRST of [E F], which is F.

The four operations FIRST, LAST, BUTFIRST, and BUTLAST are used for extracting pieces of lists. To combine lists into more complex lists, we have the Logo operation LIST. LIST takes a collection of inputs and outputs a list whose items are those inputs. For example, if there are three inputs, the word XX, the word YY, and the list [A B C], LIST outputs the list of those three items:

```
PRINT (LIST "XX "YY [A B C])
XX YY [A B C]
```

If, instead of the word XX and the word YY, you use the lists [XX] and [YY], then LIST outputs a list whose first and second items are those lists:

```
PRINT (LIST [XX] [YY] [A B C])
[XX] [YY] [A B C]
```

As another example, you can construct the list TRY used above as

```
(LIST [A B C] "D [E F])
```

or a more complex list, such as the U.S. Government list of the previous section as:

```
MAKE "EX [EXECUTIVE [PRESIDENT VICE-PRESIDENT]]
MAKE "LEG [LEGISLATIVE [SENATE HOUSE]]
MAKE "JUD [JUDICIAL [COURT]]
MAKE "GOVT (LIST :EX :LEG :JUD)
```

Another way to combine items into lists is by means of the FPUT operation. FPUT takes two inputs, of which the second must be a list. It puts its first input at the beginning of its second input, that is, it outputs a list whose FIRST is the first input and whose BUTFIRST is the second input:

```
PRINT FPUT "A [D E F]
A D E F
PRINT FPUT [A] [D E F]
[A] D E F
PRINT FPUT [A B C] [D E F]
[A B C] D E F
```

LPUT is similar to FPUT, except that it installs its first input as the *last* item in the list:

```
PRINT LPUT "A [D E F]
D E F A
PRINT LPUT [A] [D E F]
D E F [A]
PRINT LPUT [A B C] [D E F]
D E F [A B C]
```

The Logo operation SENTENCE, which we previously used to combine words into lists, can also be used with more complex lists. If SENTENCE is given a number of lists as inputs, it combines all of the elements of these lists into a single list:


```
PRINT SENTENCE [A [B C]] [D E F]
A [B C] D E F
```

Be careful about the distinction between LIST and SENTENCE. Compare the example just above with

```
PRINT LIST [A [B C]] [D E F]
[A [B C]] [D E F]
```

which results in a list of two lists. In general, LIST outputs a list of its inputs while SENTENCE combines all the *items* in its inputs into a single list.

This description of SENTENCE makes sense only when all of the inputs to SENTENCE are themselves lists. In order to make this consistent with our previous definition of SENTENCE for combining words into lists we extend the definition as follows: if one of the inputs to SENTENCE is a word, then you replace that word by the one-item list containing that word, and then apply the definition of SENTENCE given above.² For example:

```
(SENTENCE "A "B "C )
```

gives the same result as

```
(SENTENCE [A] [B] [C])
```

which is the list [A B C].

```
SENTENCE "A [B [C D]]
```

gives the same result as

```
SENTENCE [A] [B [C D]]
```

which is the list [A B [C D]]. In general, SENTENCE and LIST give the same result if all the inputs are words. SENTENCE :X :Y gives the same result as FPUT :X :Y if :X is a word and :Y is a list.

² When you are interested only in combining words into lists to be printed (as in most elementary Logo programs), SENTENCE is the only operation you need for constructing lists. However, when you are interested in using lists as hierarchical data structures, you need the finer control provided by FPUT, LPUT, and LIST. For example, it is always true that :X is the first item of FPUT :X :Y. But this is not the case with SENTENCE. For instance, if :X is [A B C] and :Y is [D E], then the first item of SENTENCE :X :Y is the word A.

9.1.2. Example: association lists

One particularly simple form of list is a list of pairs, which can be used to represent simple tables in which values are associated to things:

Type as one line. See page 9.—

```
MAKE "TABLE1 [ [COLOR PURPLE]
                [SIZE HUGE]
                [WEIGHT [1 TON]] ]
```

Such a list of pairs is called an *association list*. The first item in each pair is referred to as the *key* and second item is the corresponding *value*. The most important function for operating on tables represented as association lists is LOOKUP, which outputs the value corresponding to a given key:

```
PRINT LOOKUP "SIZE :TABLE1
HUGE
```

LOOKUP is implemented by means of an auxiliary function called ENTRY, which outputs the pair in which the key occurs, or outputs the empty list if there is no such pair in the table. LOOKUP then outputs the second item in the ENTRY, or signals an error if the key was not found. ENTRY is implemented by scanning down the list in the usual fashion:³

```
TO ENTRY :KEY :TABLE
IF :TABLE = [] [OUTPUT []]
IF :KEY = (FIRST FIRST :TABLE) [OUTPUT (FIRST :TABLE)]
OUTPUT ENTRY :KEY (BUTFIRST :TABLE)
END
```

LOOKUP is implemented as

```
TO LOOKUP :KEY :TABLE
MAKE "PAIR ENTRY :KEY :TABLE
IF NOT (:PAIR = []) [OUTPUT LAST :PAIR]
PRINT [ERROR: KEY NOT IN TABLE]
THROW "TOPLEVEL
END
```

The THROW "TOPLEVEL construct used by LOOKUP aborts execution of a Logo program and returns command to the user. It is used here to abort execution of the program when LOOKUP signals an error. See the technical manual for more information on using THROW to abort procedure execution.

³ This is very similar to the MEMBER? procedure on page 136.

Another use for association lists that arises in symbol manipulation is for substituting values from a table. The following SUBST procedure takes a list and a table as inputs. For each item in the list that is a key in the table, it replaces the key by the corresponding value. For example, with TABLE1 as above, you would have:

Type as one line. See page 9. →

```
PRINT SUBST [HE IS COLOR AND WEIGHS WEIGHT]
           :TABLE1
HE IS PURPLE AND WEIGHS [1 TON]
```

To define SUBST, we'll begin by writing a procedure SUBST.ITEM that takes an item and a table as input. If the item is a key in the table then SUBST.ITEM outputs the associated value. Otherwise it outputs the original item. Notice that this is almost the same as LOOKUP except that it returns the original item instead of signalling an error if the item is not in the table.

```
TO SUBST.ITEM :ITEM :TABLE
MAKE "SUBST.PAIR (ENTRY :ITEM :TABLE)
IF :SUBST.PAIR = [] [OUTPUT :ITEM]
OUTPUT LAST :SUBST.PAIR
END
```

The SUBST procedure itself is implemented by performing SUBST.ITEM on each item in the list, and outputting the list of the results:

Type as one line. See page 9. →

```
TO SUBST :LIST :TABLE
IF :LIST = [] [OUTPUT []]
OUTPUT FPUT (SUBST.ITEM (FIRST :LIST) :TABLE)
           (SUBST (BUTFIRST :LIST) :TABLE)
END
```

Properties

One way to think of an association list is as a collection of the attributes, or "properties," of some object:

Type as one line. See page 9. →

```
MAKE "SUPERGRAPE
    [[COLOR PURPLE] [SIZE HUGE] [WEIGHT [1 TON]]]
```

These attributes can be recovered by using the LOOKUP procedure given above. Alternatively, we can forget about the list of pairs, think in terms of a procedure PPROP for ("put property") that associates a given property value to a given symbol. For example,

```
PPROP "SUPER.GRAPE "COLOR "PURPLE
```

would associate to the symbol SUPER.GRAPE, a COLOR property whose value is PURPLE. A corresponding procedure GPROP ("get property") is used to retrieve a property value, so that, for example,

```
GPROP "SUPER.GRAPE "COLOR
```

would return PURPLE. (GPROP is usually defined to return the empty list if the symbol has no property with the given name.) A typical program that uses properties to manage information could contain a line such as

Type as one line. See page 9.—

```
PRINT (SENTENCE [THE COLOR OF]
              :ITEM
              [IS]
              (GPROP :ITEM "COLOR ))
```

PPROP and GPROP are readily implemented in terms of association lists, but in some applications, it is better to use other methods for representing properties. In particular, if there are many attributes in a table, performing a LOOKUP will be slow, due to the need to scan a long list. Apple Logo includes PPROP and GPROP as primitives, so that you can avoid the need to use long lists to implement tables. In addition, there are primitives REMPPROP, which takes a symbol and a property name as inputs and removes the corresponding property from the symbol, and PLIST, which takes a symbol as input and returns a list of that symbol's properties, together with their corresponding values.

9.2. Programs as Data

One important kind of hierarchical structure that arises in programming is the structure of a program itself. A Logo procedure can be thought of as a list of lines each of which is a list of words. Using Logo lists, you can write *programs that manipulate other programs*. The basic Logo primitives that enable you to do this are RUN, which executes a list as a Logo command line; DEFINE, which constructs a procedure from list data; and TEXT, which outputs the representation of a procedure. This section explains how these operations work in the context of an extended example—increasing the capabilities of the simple INSTANT program that was introduced in section 7.2.1.

9.2.1. The RUN command

The Logo command RUN takes a Logo list as input, and executes the list as if the list were a command line typed at the keyboard. For example

```

RUN [PRINT [HELLO THERE]]
HELLO THERE
RUN LIST "PRINT [HELLO THERE]
HELLO THERE
MAKE "COMMAND "PRINT
MAKE "INPUT [HELLO THERE]
RUN LIST :COMMAND :INPUT
HELLO THERE

```

Example: extending the INSTANT program

Another situation in which RUN is useful is where you want to build up a list of commands to be executed later. As an example, consider the INSTANT program of section 7.2.1:

```

TO INSTANT
COMMAND
INSTANT
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F [FORWARD 10]
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "C [CLEARSCREEN]
END

```

Suppose you want to add an “undo” feature to the system. That is, typing F, L, and R at the keyboard will cause the turtle to move forward, left, and right as before. In addition, typing U will cause the turtle to undo its previous move.

You can implement the undo operation as follows. As the user of the INSTANT system gives commands by pressing keys, the INSTANT program will not only move the turtle, but will also *remember* the turtle motions that were done by saving them in a list. Then, when the user wishes to undo the last command, INSTANT will clear the screen, remove the last command from the list, and reprocess the remaining commands.⁴

To implement this strategy let's assume you store the turtle commands in a list called HISTORY. For example, if the user types F and then R, HISTORY will be

```
[ [FORWARD 10] [RIGHT 30] ]
```

⁴ There are, of course, many other ways to implement the undo operation. One advantage of the way chosen here is that it extends nicely to allow the user of the INSTANT system to define programs, as we shall see in section 9.2.2.

Notice that HISTORY is a list of lists, in which each entry is the Logo command that should be run to cause the appropriate turtle motion.

The main operation needed now is to take a turtle command and not only do it, but also add it to the HISTORY list. This can be accomplished by

```
TO RUN.AND.RECORD :ACTION
RUN :ACTION
MAKE "HISTORY (LPUT :ACTION :HISTORY)
END
```

You use LPUT to add the new command as the *last* item in HISTORY.

You now change the COMMAND procedure to RUN.AND.RECORD the appropriate response to each key:

```
TO COMMAND
MAKE "COM READCHAR
IF :COM = "F [RUN.AND.RECORD [FORWARD 10]]
IF :COM = "R [RUN.AND.RECORD [RIGHT 30]]
IF :COM = "L [RUN.AND.RECORD [LEFT 30]]
IF :COM = "C [RUN.AND.RECORD [CLEARSCREEN]]
END
```

Now, to undo the last command, you remove the last item from HISTORY, clear the screen, and run the rest of the commands:

```
TO UNDO
IF :HISTORY = [ ] [STOP]
MAKE "HISTORY BUTLAST :HISTORY
CLEARSCREEN
RUN.ALL :HISTORY
END
```

Note the first line of UNDO, which says that if the HISTORY list is empty, there is nothing to undo. Also note that with this implementation, repeatedly executing UNDO keeps removing more and more items from HISTORY, starting with the last one, the one before that, and so on.

The subprocedure RUN.ALL takes a list of commands as input and runs all the commands in the list in sequence. (Each command in the list must itself be a list.) RUN.ALL uses a recursive strategy. It RUNs the first command in the list and then processes the BUTFIRST of the list.

```

TO RUN.ALL :COMMANDS
IF :COMMANDS = [ ] [STOP]
RUN FIRST :COMMANDS
RUN.ALL (BUTFIRST :COMMANDS)
END

```

Now all you need to do is add a line to the COMMAND procedure so that pressing U causes an UNDO operation:

```

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F [RUN.AND.RECORD [FORWARD 10]]
IF :COM = "R [RUN.AND.RECORD [RIGHT 30]]
IF :COM = "L [RUN.AND.RECORD [LEFT 30]]
IF :COM = "C [RUN.AND.RECORD [CLEARSCREEN]]
IF :COM = "U [UNDO]
END

```

The complete INSTANT program now simply clears the screen and repeatedly calls COMMAND. You also need to initialize HISTORY to be empty:

```

TO SETUP
MAKE "HISTORY [ ]
CLEARSCREEN
INSTANT
END

TO INSTANT
COMMAND
INSTANT
END

```

9.2.2. The DEFINE command

In addition to using Logo list operations to generate individual command lines that can be RUN, you can also write procedures that define other procedures. This is done with the DEFINE command. DEFINE takes two inputs. This first is the name of the procedure to be defined. The second input is a list of lists organized as follows. The first sublist gives the inputs to the new procedure, and there is one additional sublist for each procedure line. For example

```

DEFINE "TRY [[X Y][PRINT :X][PRINT :Y]]
PO "TRY
TO TRY :X :Y
PRINT :X
PRINT :Y
END

```

```

DEFINE "GREET [[ ] [PRINT [HELLO]]]
PO "GREET
TO GREET
PRINT [HELLO]
END

```

Observe that if the procedure is to have no inputs (as in GREET above), the DEFINE list must include an initial empty list for the input specification. Note also that there is no END included in the list of procedure lines.

Example: another extension to INSTANT

Most of the time, of course, you use TO rather than DEFINE to create Logo procedures. DEFINE is reserved for those situations in which you want procedure definition to happen within a program. As an example of this, we'll consider another extension to the INSTANT system of section 9.2.1. This time, we'll allow the user of INSTANT to name drawings and to recall them by name. For example, we may use the letter S for saving drawings. Typing S (for "save") will cause the program ask the user for a name for the drawing. Later on, the user can ask for a previous drawing to be reshowed, say by typing P for "picture." More than one drawing can be saved at once, each with its own name.

You can implement this by having the INSTANT system save a drawing by *defining the drawing as a procedure*, using the name chosen by the user. The list of lines in the procedure is precisely the HISTORY list that you have been using to keep track of what is on the screen. Here is the procedure that implements this "learning" process:

```

TO LEARN
PRINT [WHAT DO YOU WANT TO CALL THIS PICTURE?]
MAKE "NAME (FIRST READLIST)
DEFINE :NAME (FPUT [ ] :HISTORY)
SETUP
END

```

The reason for taking the NAME of the procedure to be FIRST of READLIST is that READLIST always outputs the typed line as a list, and DEFINE needs the procedure name to be specified as a word. Also, note that the second input given to DEFINE is FPUT [] :HISTORY, since you need to include an empty input list for the procedure being defined. Also, you should make LEARN clear the screen and reinitialize HISTORY to prepare for a new drawing.

The behavior of LEARN is now:

WHAT DO YOU WANT TO CALL THIS PICTURE?
BOX

and there is now a procedure called BOX, which, when run, draws the picture that currently appears on the screen.

Now you must add a command that asks for an input line and runs it. This is accomplished by

```
TO ASK
PRINT [WHAT PICTURE DO YOU WANT TO SHOW?]
RUN.AND.RECORD READLIST
END
```

Notice that the input READLIST line is both run and recorded. Note also that *any* Logo command could be input and executed, not just a call to a procedure created by LEARN.

Finally, you need only add the appropriate lines to the COMMAND procedure so that it will recognize the characters S (for save) and P (for picture) and run the appropriate procedures.

The complete INSTANT system

Here is a complete listing of the INSTANT system developed in the preceding sections:

```
TO SETUP
MAKE "HISTORY [ ]
CLEARSCREEN
INSTANT
END

TO INSTANT
COMMAND
INSTANT
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F [RUN.AND.RECORD [FORWARD 10]]
IF :COM = "R [RUN.AND.RECORD [RIGHT 30]]
IF :COM = "L [RUN.AND.RECORD [LEFT 30]]
IF :COM = "C [RUN.AND.RECORD [CLEARSCREEN]]
IF :COM = "U UNDO
IF :COM = "S LEARN
IF :COM = "P ASK
END
```

```

TO RUN.AND.RECORD :ACTION
RUN :ACTION
MAKE "HISTORY (LPUT :ACTION :HISTORY)
END

```

```

TO UNDO
IF :HISTORY = [ ] [STOP]
MAKE "HISTORY BUTLAST :HISTORY
CLEARSCREEN
RUN.ALL :HISTORY
END

```

```

TO RUN.ALL :COMMANDS
IF :COMMANDS = [ ] [STOP]
RUN FIRST :COMMANDS
RUN.ALL (BUTFIRST :COMMANDS)
END

```

```

TO LEARN
PRINT [WHAT DO YOU WANT TO CALL THIS PICTURE?]
MAKE "NAME (FIRST READLIST)
DEFINE :NAME (FPUT [ ] :HISTORY)
SETUP
END

```

```

TO ASK
PRINT [WHAT PICTURE DO YOU WANT TO SHOW?]
RUN.AND.RECORD READLIST
END

```

There are many possible modifications and improvements to this system. For a good exercise in manipulating lists, consider the following problem. A typical HISTORY list to be assembled into a procedure might look like:

```

FORWARD 10
RIGHT 30
LEFT 30
FORWARD 10
RIGHT 30
RIGHT 30
RIGHT 30
FORWARD 10
FORWARD 10

```

It would be nice if, before the HISTORY list is made into a procedure, it could be "compressed" so that the procedure that is defined would consist of the command sequence

```
FORWARD 20
RIGHT 90
FORWARD 20
```

Write a procedure **COMPRESS** that will perform this kind of transformation on a list of turtle commands. Once you have **COMPRESS**, the **LEARN** procedure can be rewritten as:

```
TO LEARN
PRINT [WHAT DO YOU WANT TO CALL THIS PICTURE?]
MAKE "NAME (FIRST READLIST)
DEFINE :NAME (FPUT [ ] (COMPRESS :HISTORY))
SETUP
END
```

9.2.3. The **TEXT** command

In some instances, it is useful to have an “inverse operation” to **DEFINE**, that is, to be able to take a procedure that is already defined and to extract the text of the procedure so that it can be manipulated as a list. This is done with the Logo command **TEXT**, which takes a procedure name as input and outputs the text of the procedure in the same format as is used in **DEFINE**.

For example, assume that **CORNER** is defined as

```
TO CORNER :A :B
FORWARD :A
RIGHT :B
END
```

Then **TEXT** “**CORNER** is the list

```
[[A B] [FORWARD :A] [RIGHT :B]]
```

Using **TEXT**, you can write procedures that examine and manipulate other procedures.

9.2.4. Adding new programming constructs

The ability to use list operations to construct lists, and then to **RUN** these lists as commands, allows you to add new programming constructs to the basic Logo language. For instance, suppose you would like to have a **WHILE** command that can be used to keep repeating something over and over as long as some condition is true, as in:

```
WHILE [XCOR < 20] [FORWARD 1]
```

Logo does not include **WHILE** as a primitive command. But you can use **RUN** to define your own **WHILE** command as a procedure that takes two lists as inputs. The first list specifies a

condition to be tested, and the second list specifies an action to be repeated over and over as long as the condition remains true. The WHILE procedure first tests if the condition is true by RUNNING the condition list. If the result is true, the WHILE procedure RUNs the action list. This sequence is repeated over and over:

```
TO WHILE :CONDITION :ACTION
IF NOT (RUN :CONDITION) [STOP]
RUN :ACTION
WHILE :CONDITION :ACTION
END
```

As a more complex example, you can implement a FOR procedure that works as follows:

```
FOR [COUNT 1 5] [PRINT :COUNT * :COUNT]
1
4
9
16
25
```

The FOR procedure takes two lists as inputs. The first list is a “FOR list” that specifies a loop variable together with two integer initial and final values. The second input specifies an action that should be executed for all values of the loop variable between the initial and final values. To implement FOR, you extract from the FOR list the name of the variable and the initial and final values, and pass these on to a subprocedure FOR.LOOP, which does the actual work of looping. It is convenient to write separate, short procedures to extract the parts of the FOR list:

```
TO VAR :FLIST
OUTPUT FIRST :FLIST
END

TO INITIAL :FLIST
OUTPUT FIRST BUTFIRST :FLIST
END

TO FINAL :FLIST
OUTPUT LAST :FLIST
END
```

Then the FOR procedure is written as:

Type as one line. See page 9.—

```
TO FOR :FLIST :ACTION
FOR.LOOP (VAR :FLIST)
  (INITIAL :FLIST)
  (FINAL :FLIST)
  :ACTION
END
```

The FOR.LOOP procedure takes as inputs the variable, the initial and final values, and the action to be RUN. It uses MAKE to set the variable to the initial value and RUNs the action. Then it repeats the sequence, with the initial value incremented by 1. As a stop rule, FOR.LOOP tests to see whether the initial value has become greater than the final value. Here is the procedure:

```
TO FOR.LOOP :VAR :INITIAL :FINAL :ACTION
IF :INITIAL > :FINAL [STOP]
MAKE :VAR :INITIAL
RUN :ACTION
FOR.LOOP :VAR (:INITIAL + 1) :FINAL :ACTION
END
```

Note that in the second line of FOR.LOOP, you say MAKE :VAR, rather than MAKE "VAR, because the name of the variable being set is the value associated with VAR, rather than the literal word VAR. For example, in executing the command

```
FOR [COUNT 1 5] [PRINT :COUNT * :COUNT]
```

the value of VAR is the word COUNT, and COUNT is the variable you want to set using the MAKE command. Compare the "tricky use of MAKE" shown on page 87.

9.3. More Projects Using Lists

This section presents two open-ended projects that make use of Logo's capability to manipulate lists. The first project is a highly simplified version of a famous program called DOCTOR that was developed in the early 1960s by Joseph Weizenbaum. The Logo implementation here uses the association list mechanism that was explained in section 9.1.2. The second program is a Logo version of a well-known computer "learning" program called ANIMAL, which demonstrates how to use hierarchical list structures to keep track of information.

9.3.1. Example: the DOCTOR Program

The DOCTOR program engages in a "conversation" with the computer user, in which it mimics a nondirective style of psychotherapy. Here is a sample conversation:⁵

⁵ In order to simplify the program all punctuation has been omitted. As usual, the computer's typeout has been italicized to distinguish it from the user's responses.

DOCTOR
WHAT SEEMS TO BE THE PROBLEM

EVERYONE HATES ME
YOU SAY EVERYONE HATES YOU

WHENEVER I ENTER A ROOM NO ONE WILL LOOK ME IN
 THE EYE
MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS

THEY LAUGH AT ME BEHIND MY BACK
*YOU SEEM TO THINK THEY LAUGH AT YOU BEHIND
 YOUR BACK*

EVEN MY MOTHER HATES ME
YOU FEEL THAT EVEN YOUR MOTHER HATES YOU

EVEN WHEN I WAS A BABY SHE SAID I WAS ALWAYS
 BAD
PLEASE CONTINUE

How the DOCTOR program works

Although at first glance, the DOCTOR program seems to understand and reply to the user's remarks, in reality it is doing nothing of the sort. In fact, the program has two simple methods for generating a response. The first method is to ignore what the user types and simply respond with some sort of hedge like PLEASE CONTINUE or MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS. The second method involves taking the user's reply, changing some common words like "I," "me," and "am" to the corresponding second-person words and appending the transformed response to some qualifying phrase such as YOU SAY or YOU SEEM TO THINK. The program chooses one of these methods at random for each response.

We'll examine these two methods in turn. The first is very simple. What the program prints is just a phrase picked at random from a suitable list of hedges such as

Type as one line. See page 9.—

```
MAKE "HEDGES
  [ [PLEASE GO ON]
    [PLEASE CONTINUE]
    [MANY PEOPLE HAVE THE SAME SORTS OF
      FEELINGS] ]
```

Then the part of the program that implements the first method is just⁶

⁶ We use here the PICKRANDOM procedure (page 101).

```

TO HEDGE
PRINT PICKRANDOM :HEDGES
END

```

The second method is more complicated. You must take the user's typed-in response, change the "I" words to the corresponding "you" words and append this to a randomly selected qualifier. To perform the "I-you" change, you can use the SUBST procedure on page 148, where the substitution TABLE of pairs is made up of first-person pronouns and their second-person counterparts:

Type as one line. See page 9. →

```

MAKE "PRONOUNS
  [[I YOU] [ME YOU] [MY YOUR] [AM ARE]]

TO CHANGE.PERSON :PHRASE
OUTPUT SUBST :PHRASE :PRONOUNS
END

```

So if the collection of qualifiers is given by

Type as one line. See page 9. →

```

MAKE "QUALIFY
  [[YOU SEEM TO THINK]
  [YOU FEEL THAT]
  [YOU SAY]]

```

then the second type of response to the user's input is generated by

Type as one line. See page 9. →

```

TO RESPOND :USER.INPUT
PRINT SE (PICKRANDOM :QUALIFY)
  (CHANGE.PERSON :USER.INPUT)
END

```

Now you can put both methods together. You can select between the methods at random, using the test IF (RANDOM 2) = 0 to generate TRUE or FALSE with equal chances. You can also terminate the conversation if the user types GOODBYE:

Type as one line. See page 9. →

```

TO DOCTOR.LOOP
MAKE "USER.INPUT REQUEST
IF :USER.INPUT = [GOODBYE]
  [PRINT [COME SEE ME AGAIN] STOP]
IF (RANDOM 2) = 0 [HEDGE] [RESPOND :USER.INPUT]
DOCTOR.LOOP
END

```

All that is missing now is a procedure DOCTOR to start things going. This should initialize the lists QUALIFY,

HEDGES, and PRONOUNS used above, print an opening remark, and call DOCTOR.LOOP:

Type as one line. See page 9. →

TO DOCTOR
MAKE "QUALIFY
[[YOU SEEM TO THINK]
[YOU FEEL THAT]
[YOU SAY]]

Type as one line. See page 9. →

MAKE "HEDGES
[[PLEASE GO ON]
[PLEASE CONTINUE]
[MANY PEOPLE HAVE THE SAME SORTS OF
FEELINGS]]

Type as one line. See page 9. →

MAKE "PRONOUNS
[[I YOU]
[ME YOU] [MY YOUR] [AM ARE]]
PRINT [WHAT SEEMS TO BE THE PROBLEM]
DOCTOR.LOOP
END

Extending the program

The previous program is only a simple sketch. One immediate extension you'll want to make is to increase its repertoire of HEDGES and QUALIFY, so that the responses are more varied. Another idea is to upgrade the RESPOND procedure not only to change first-person words to second person, but also second person to first. For instance, if the user types

YOU ARE NOT BEING VERY HELPFUL TO ME

the program should respond with something like

YOU FEEL THAT I AM NOT BEING VERY HELPFUL TO YOU

Another idea is this. Every so often, the program should save away the user's response. Then, a few exchanges later, the program could say something like "Earlier you said that . . ." Still other ideas are to have the program select special responses, when the user mentions certain words, like "computer."

By including more and more of these features, you can make the program's conversations quite elaborate. The responses of Weizenbaum's original DOCTOR program have been occasionally mistaken for those of a real person, and this has led some people to advocate using such programs in the treatment of psychiatric patients. Others, including Weizenbaum, maintain that this would be extremely unethical. For a further discussion of these issues see Weizenbaum's book [15].

9.3.2. Example: The ANIMAL program

ANIMAL is a well-known computer program that asks the user to think of an animal and then tries to guess what animal it is by asking yes-or-no questions. Here is a sample session with the program:

```
ANIMAL
THINK OF AN ANIMAL. I WILL
TRY TO GUESS IT BY ASKING QUESTIONS.
DOES IT HAVE LEGS?
YES
IS IT A CAT?
YES
LOOK HOW SMART I AM!
LET'S TRY AGAIN. . .
THINK OF AN ANIMAL. I WILL
TRY TO GUESS IT BY ASKING QUESTIONS.
DOES IT HAVE LEGS?
NO
DOES IT CRAWL?
YES
IS IT A SNAKE?
YES
LOOK HOW SMART I AM!
LET'S TRY AGAIN. . .
.
.
.
```

The cleverness of the program is that it learns from its mistakes. Here is what happens when it guesses incorrectly:

```
DOES IT HAVE LEGS?
NO
DOES IT CRAWL?
YES
IS IT A SNAKE?
NO
OH WELL, I WAS WRONG. WHAT WAS IT?
EARTHWORM
PLEASE TYPE IN A QUESTION WHOSE ANSWER
IS YES FOR AN EARTHWORM AND
NO FOR A SNAKE
DOES IT LIVE UNDERGROUND?
```

LET'S TRY AGAIN. . .

.
.
.

The next time the program runs across this situation it will behave like this:

DOES IT HAVE LEGS?

NO

DOES IT CRAWL?

YES

DOES IT LIVE UNDERGROUND?

.
.
.

So the program becomes smarter and smarter as it is used more and more.

How the ANIMAL program works

The key to the program is its knowledge structure. This can be thought of as a tree, as shown in figure 9.2. The tree is made up of "nodes," where each node consists of a QUESTION to ask, a YES.BRANCH to follow if the answer to the question is yes, and a NO.BRANCH to follow if the answer is no.

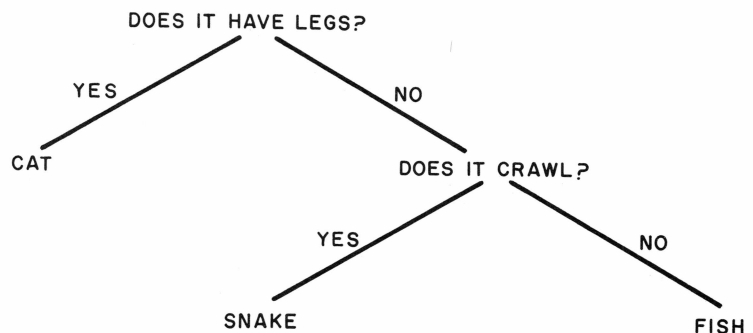


Figure 9.2: Knowledge tree for the ANIMAL program.

So the basic operation of the program is to begin at the top node of the tree and work its way down, following the YES.BRANCH or the NO.BRANCH according to the answer to

the QUESTION. If the program reaches a node that consists of only a single item, it guesses that as the animal.

When the program guesses incorrectly, it “gets smarter” by expanding the tree. It asks the user for the correct response and for a question that distinguishes the correct response from the incorrect response. It then replaces the old single-item node by a new node made up of the user’s question, the correct response as the YES.BRANCH and the old incorrect response as the NO.BRANCH.⁷ For example, to learn the difference between a snake and an earthworm, the program expands the tree, replacing the SNAKE node by a node whose QUESTION is DOES IT LIVE UNDERGROUND?, whose YES.BRANCH is EARTHWORM, and whose NO.BRANCH is SNAKE.

That’s all there is to it.

Using lists

The ANIMAL program can be conveniently written in Logo, because lists are just the right tool for representing the knowledge tree. You can think of the tree as a list called KNOWLEDGE that has three elements: a QUESTION, a YES.BRANCH, and a NO.BRANCH. Of course YES.BRANCH and NO.BRANCH may themselves be lists that have the same structure. And so you have sublists and sublists, until you finally reach branches that are words, which give the actual animals to be guessed.

Here is a Logo list that represents the tree shown in figure 9.2:

```
[ [DOES IT HAVE LEGS?]
  CAT
  [ [DOES IT CRAWL?]
    SNAKE
    FISH]]
```

When snake is distinguished from earthworm, the list becomes

```
[ [DOES IT HAVE LEGS?]
  CAT
  [ [DOES IT CRAWL?]
    [ [DOES IT LIVE UNDERGROUND?]
      EARTHWORM
      SNAKE]
    FISH] ]
```

⁷ Of course, if the user types in *wrong* information, the program will get stupider instead of smarter. Also, the program we shall describe below does not check for *inconsistent* responses on the part of the user. Extending the program to do so is a good project.

With the program's knowledge structured in this way, you can extract the QUESTION, YES.BRANCH, and NO.BRANCH parts of a given node by using the following procedures:

```
TO QUESTION :NODE
  OUTPUT FIRST :NODE
END
```

```
TO YES.BRANCH :NODE
  OUTPUT FIRST (BUTFIRST :NODE)
END
```

```
TO NO.BRANCH :NODE
  OUTPUT LAST :NODE
END
```

The main procedure

Here is the procedure that starts the program:

```
TO ANIMAL
  PRINT [THINK OF AN ANIMAL. I WILL]
  PRINT [TRY TO GUESS IT BY ASKING QUESTIONS]
  CHOOSE.BRANCH :KNOWLEDGE
  PRINT [LET'S TRY AGAIN. . .]
  ANIMAL
END
```

It prints the instructions, does the guessing, and continues this over and over. The real work is done by the CHOOSE.BRANCH procedure, which is meant to be called with a node as input. It is initially called with the node that is the entire KNOWLEDGE list of the program:

```
TO CHOOSE.BRANCH :NODE
  IF (WORDP :NODE) [GUESS :NODE STOP]
  MAKE "RESPONSE ASK.YES.OR.NO (QUESTION :NODE)
  IF :RESPONSE = [YES]
    [CHOOSE.BRANCH (YES.BRANCH :NODE) STOP]
  CHOOSE.BRANCH (NO.BRANCH :NODE)
END
```

Type as one line. See page 9.→

CHOOSE.BRANCH implements precisely the technique explained above. It asks the question associated with the node and then continues with the YES.BRANCH or the NO.BRANCH according to the result of the question. When it reaches a node that is a single word, it uses that as its guess. (The GUESS procedure, which actually makes the guess, is discussed below.) Notice how the "continues with . . ." part of the strategy is

implemented by a CHOOSE.BRANCH calling itself recursively using the appropriate branch as the new node.

Asking questions

The following procedure is used to ask a yes-or-no question. It takes the question as input and returns either [YES] or [NO].

```
TO ASK.YES.OR.NO :QUESTION
PRINT :QUESTION
MAKE "INPUT READLIST
IF :INPUT = [YES] [OUTPUT [YES]]
IF :INPUT = [NO] [OUTPUT [NO]]
PRINT [PLEASE TYPE "YES" OR "NO"]
OUTPUT ASK.YES.OR.NO :QUESTION
END
```

If the user responds with something other than YES or NO, the procedure repeats the question, using the same "try again" method as with the READNUMBER procedure on page 100.

"A" or "an"

One nicety that the program must handle when making guesses is to distinguish between animal names that begin with vowels and those that do not. So, if the guess is "snake" the program should ask "Is it *a* snake?" while, if the guess is "earthworm" the program should ask "Is it *an* earthworm?" The following procedure helps to do this. It takes a word as input and outputs a sentence consisting of the word preceded by "a" or "an" as appropriate:

```
TO ADD.A.OR.AN :WORD
TEST MEMBERP (FIRST :WORD) [A E I O U]
IFTRUE [OUTPUT SENTENCE "AN :WORD]
IFFALSE [OUTPUT SENTENCE "A :WORD]
END
```

Making a guess

When CHOOSE.BRANCH reaches a node with only a single animal, it calls the GUESS procedure with that animal as input.

```
TO GUESS :ANIMAL
MAKE "FINAL.QUESTION
(SE [IS IT] (ADD.A.OR.AN :ANIMAL) [?])
MAKE "RESPONSE ASK.YES.OR.NO :FINAL.QUESTION
```

Type as one line. See page 9. —

Type as one line. See page 9. →

```
IF :RESPONSE = [YES]
  [PRINT [LOOK HOW SMART I AM!] STOP]
GET.SMARTER :ANIMAL
END
```

GUESS first formulates the appropriate “Is it (a or an) . . . ?” question and gets the response. If the guess is correct, the program brags about how smart it is and stops, returning eventually to the ANIMAL procedure, which starts the next round. If the guess is wrong, the program must grow smarter.

Getting smarter

Getting smarter consists, first of all, of asking the user for the right animal and for a question that distinguishes the right animal from the wrong one. Observe how the “a or an” choice is needed to construct the request for a question.

Type as one line. See page 9. →

```
TO GET.SMARTER :WRONG.ANSWER
  PRINT [OH WELL, I WAS WRONG. WHAT WAS IT?]
  MAKE "RIGHT.ANSWER (LAST READLIST)
  PRINT [PLEASE TYPE IN A QUESTION WHOSE ANSWER]
  PRINT (SE [IS YES FOR]
    (ADD.A.OR.AN :RIGHT.ANSWER)
    [AND])
  PRINT (SE [NO FOR] (ADD.A.OR.AN :WRONG.ANSWER))
  MAKE "QUESTION READLIST
  EXTEND.KNOWLEDGE :QUESTION
    :RIGHT.ANSWER
    :WRONG.ANSWER
END
```

Type as one line. See page 9. →

Once the new question and the two answers are in hand, the program proceeds to extend its knowledge. This is accomplished by resetting the list KNOWLEDGE to the result of starting with KNOWLEDGE and replacing the node consisting of just the old answer with a list formed from the question, the new animal as the YES.BRANCH, and the old answer as the NO.BRANCH.

Type as one line. See page 9. →

```
TO EXTEND.KNOWLEDGE :NEW.QUESTION
  :YES.ANSWER
  :NO.ANSWER
```

Type as one line. See page 9. →

```
MAKE "KNOWLEDGE
  REPLACE :KNOWLEDGE
    :NO.ANSWER
    (LIST :NEW.QUESTION
      :YES.ANSWER
      :NO.ANSWER)
END
```

Finally, there is the procedure that does the actual replacement. This takes as inputs

- A list that represents a tree of QUESTION—YES.BRANCH—NO.BRANCH nodes
- A node to be replaced
- The thing to replace it with

The output of REPLACE is a copy of the tree with the old node replaced by the designated replacement.

Type as one line. See page 9. →

```
TO REPLACE :TREE :NODE :REPLACEMENT
  IF :TREE = :NODE [OUTPUT :REPLACEMENT]
  IF WORDP :TREE [OUTPUT :TREE]
  OUTPUT (LIST QUESTION :TREE
    REPLACE (YES.BRANCH :TREE)
      :NODE
      :REPLACEMENT
    REPLACE (NO.BRANCH :TREE)
      :NODE
      :REPLACEMENT)
END
```

REPLACE is the most difficult procedure in the ANIMAL program. It uses a recursive strategy somewhat as in the SUBST procedure (section 9.1.2) but more complicated. The idea is that if the tree itself is the node to replace, you output the replacement. Otherwise, the new tree should be formed from the original tree's QUESTION, together with the result of performing the replacement recursively in the YES.BRANCH and the NO.BRANCH. This reduces the substitution to operations on smaller and smaller subtrees of the original. Finally, when you reduce to nodes that are individual words, you should output the words themselves.

Running the program

Figure 9.3 shows the structure of procedure calls for the entire ANIMAL program.

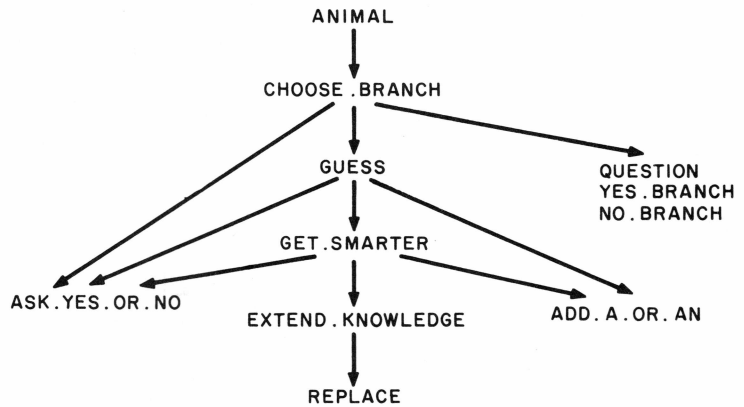


Figure 9.3: Structure of procedure calls in the ANIMAL program.

To run the program, you initialize the KNOWLEDGE list to any animal you like, and run the ANIMAL procedure. For example,

```
MAKE "KNOWLEDGE "FISH
ANIMAL
```

When you are done with your session, you can save your workspace, which will include the final value of the KNOWLEDGE tree. Thus when you load the program in to play with it the next time, the wisdom gained during previous sessions will not be lost.

Modifications

ANIMAL is a lot of fun to use as it is, especially as you begin to input funny animals and outrageous questions. There are also many good projects that arise from extending the program's capabilities. One idea is to vary the responses that the program gives. Another is to have the program request not a literal yes-or-no question, but rather a fact that distinguishes between the two animals. Then the program can use this fact to generate its own question. This requires that you build in some knowledge about grammar, somewhat like the capabilities in the DOCTOR program. Finally—and this is really open-ended—you can try to make the program smart enough to distinguish legitimate responses from wisecracks. Here is a sample dialogue with a more sophisticated ANIMAL program:⁸

⁸ The dialogue given here is with a program by B. Greenberg [8].

DOES IT HAVE HORNS?

YES

IS IT A BUFFALO?

NO

*WELL, I'M NOT TOO SHARP TODAY. I GIVE UP.
JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?
A GAZELLE*

*TELL ME SOMETHING ABOUT A GAZELLE THAT IS NOT
TRUE ABOUT A BUFFALO
A GAZELLE IS GRACEFUL*

*THAT WAS FUN. WANT TO TRY AGAIN?
MAYBE*

*HEY, CAN YOU GIVE ME A YES OR NO ANSWER
YES*

DOES IT HAVE HORNS?

NO

IS IT A BUTTERFLY?

NO

*WELL, I'M NOT TOO SHARP TODAY. I GIVE UP.
JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?
A PIG*

*TELL ME SOMETHING ABOUT A PIG THAT IS NOT TRUE
ABOUT A BUTTERFLY
A PIG IS SLOVENLY*

*THAT WAS FUN. WANT TO TRY AGAIN?
YES*

DOES IT HAVE HORNS?

NO

IS THIS ANIMAL SLOVENLY?

YES

IS IT A PIG?

YES

*HEY, HEY, I SURE AM CLEVER, HUH?
THAT WAS FUN. WANT TO TRY AGAIN?
YES*

DOES IT HAVE HORNS

YES

IS THIS ANIMAL GRACEFUL?

YOU ARE NOT GRACEFUL

*HEY CAN YOU GIVE ME A YES OR NO ANSWER?
NO*

IS IT A BUFFALO?

NO

WELL, I'M NOT TOO SHARP TODAY. I GIVE UP.

JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?

BULL

*TELL ME SOMETHING ABOUT A BULL THAT IS NOT TRUE
ABOUT A BUFFALO*

WHY DON'T YOU TELL ME SOMETHING, YOU
ELECTRONIC MORON?

AW, BE SERIOUS. I ASKED YOU A REAL QUESTION

IT WOULD MARRY A COW

.

.

.



CHAPTER 10

Glossary of Logo Primitive Commands

This chapter lists the primitive commands included in the Apple Logo system together with their abbreviations and examples of how many of them are used. As in the rest of this book, when we wish to emphasize the distinction between what the user types and what the computer responds, we have printed the latter in *italics*.

10.1 Graphics Commands

These are Logo's commands for drawing and using the turtle and the graphics screen.

BACK Abbreviated BK
Example:

BACK 100
{turtle moves backward 100 units}

Takes one number as input and moves the turtle that many units in the direction opposite from that in which the turtle is facing. Draws a line if the pen is down.

BACKGROUND Takes no inputs. Returns a number between 0 and 6 which specifies the color of the background. The correspondence between numbers and colors is given in section 1.4.1.

CLEAN Takes no inputs. Clears the graphic screen. Does not change the turtle's position.

CLEARSCREEN Abbreviated CS
Takes no inputs. Clears the graphics screen and homes and shows the turtle.

DOT Example:

DOT [20 100]
{a dot appears on the screen at position (20,100)}

Takes as input a list specifying a screen position and places a dot (in the current pencolor) at that position. Does not move the turtle.

FENCE	Takes no inputs. Cause Logo to signal the error TURTLE OUT OF BOUNDS in response to any attempt to move the turtle past the screen boundaries. Compare WRAP and WINDOW.
FORWARD	Abbreviated FD Example: FORWARD 100 {turtle moves forward 100 units} Takes one numeric input. Moves the turtle the designated number of units in the direction in which it is facing. Draws a line if the pen is down.
FULLSCREEN	In graphics mode, gives full graphics screen. Complementary to SPLITSCREEN and TEXTSCREEN. Equivalent to interrupt character CTRL-L.
HEADING	Example: SETHEADING HEADING + 10 {rotates the turtle 10 degrees clockwise} Takes no inputs. Outputs the turtle's heading as a decimal number between 0 and 360.
HIDETURTLE	Abbreviated HT Takes no inputs. Makes the turtle pointer disappear.
HOME	Takes no inputs. Moves the turtle to the center of the screen, pointing straight up.
LEFT	Abbreviated LT Example: LEFT 90 {turtle rotates 90 degrees counterclockwise} Takes one numeric input. Rotates the turtle that many degrees counterclockwise.
PEN	Takes no inputs. Outputs a list of two elements that describes the state of the pen. The first element is either PENUP, PENDOWN, PENERASE or PENREVERSE. The second element is a number that specifies the current pen color as described in section 1.4.1.
PENCOLOR	Abbreviated PC Takes no inputs. Outputs a number between 0 and 5 that specifies the current pen color as described in section 1.4.1.

- PENDOWN** Abbreviated PD
Takes no inputs. Causes the turtle to leave a trail when it moves.
- PENERASE** Abbreviated PE
Takes no inputs. Puts the turtle's eraser down, so that when the turtle moves, any point it passes over will be erased (that is, changed to the background color).
- PENREVERSE** Abbreviated PX
Takes no inputs. Changes the turtle's pen so that it "reverses" any point that it passes over.
- PENUP** Abbreviated PU
Takes no inputs. Causes the turtle to move without leaving a trail.
- POS** Takes no inputs. Outputs a list of two numbers that specifies the turtle's current position in x,y coordinates.
- RIGHT** Abbreviated RT
Example:

RIGHT 45
{turtle rotates 45 degrees clockwise}

Takes one numeric input. Rotates the turtle that many degrees clockwise.
- SETBG** Takes a number 0-6 as input and changes the background to the specified color. The correspondence between numbers and colors is given in section 1.4.1.
- SETHEADING** Abbreviated SETH
Example:

SETHEADING 180
{turtle now faces straight down}

Takes one numeric input. Rotates the turtle to point in the direction specified. The input is interpreted as a number in degrees. Zero is straight up, with heading increasing clockwise.
- SETPC** Example:

SETPC 3
{turtle will now draw in violet}

Takes one input, which must be an integer from 0 to 5. This determines the color of the lines that the turtle will draw, as described in section 1.4.1.

SETPEN	Takes a two-element list as input and sets the turtle's pen accordingly. The format of the list is as described under PEN.
SETPOS	Takes a list of two numbers as input and moves the turtle to the specified point. Draws a line if the pen is down.
SETX	Takes one numeric input and moves the turtle horizontally to the specified coordinate. Draws a line if the pen is down.
SETY	Takes one numeric input and moves the turtle vertically to the specified coordinate. Draws a line if the pen is down.
SHOWNP	Takes no inputs. Outputs TRUE if the turtle is currently shown, FALSE otherwise.
SHOWTURTLE	Abbreviated ST Takes no inputs. Makes the turtle pointer appear.
SPLITSCREEN	Takes no inputs. In graphics mode, gives mixed text/graphics screen. Complementary to FULLSCREEN and TEXTSCREEN. Equivalent to interrupt character CTRL-S.
TEXTSCREEN	Takes no inputs. Hides the graphics screen so that the entire screen can be used for text. Complementary to SPLITSCREEN and FULLSCREEN. Equivalent to interrupt character CTRL-T.
TOWARDS	Example: SETHEADING TOWARDS [100 50] {turtle is now facing point (100,50)} Takes a list of two numbers as input. These are interpreted as the <i>x</i> and <i>y</i> coordinates of a point on the screen. TOWARDS outputs the heading from the turtle to the point.
WINDOW	Takes no inputs. Makes the graphics screen act like a window onto a much larger turtle field. Thus the turtle can move past the edge of the screen, but only the portion of the turtle's trail that lies in the window will be shown on the screen. See also WRAP and FENCE.
WRAP	Takes no inputs. Causes subsequent turtle commands to "wraparound" in response to attempts to move the turtle past the screen borders. That is to say, a turtle path that goes off the top of the screen will continue from the bottom of the screen at the same horizontal position, a path that goes off the right edge of the screen will continue from the left edge of the screen at the same vertical position, and so on. This wrap-around effect is Logo's default behavior. Alternative modes are selected by FENCE and WINDOW.

XCOR Example:

```
SETX XCOR + 10
{moves the turtle 10 units to the right}
```

Takes no inputs. Outputs the turtle's x coordinate as a decimal number.

YCOR Takes no inputs. Outputs the turtle's y coordinate as a decimal number.

10.2. Numeric Operations

These are Logo's built in facilities for performing operations with numbers.

+ Example:

```
PRINT 5 + 2.5
7.5
```

Takes two numbers as inputs, and outputs their sum.

- Example:

```
PRINT 5 - 2
3
PRINT 1 + (-2)
-1
```

With two numeric inputs, outputs their difference. With one numeric input, outputs its negative if there is no space between the minus sign and its input. See reference manual, Appendix I, Parsing.

***** Example:

```
PRINT 5 * 2
10
```

Takes two numeric inputs, and outputs their product.

/ Example:

```
PRINT 5 / 2
2.5
PRINT 6 / 2
3.
```

Outputs its first input divided by its second. Always outputs a decimal value.

ARCTAN Takes two numeric inputs, x and y , and outputs an angle whose arctangent is x/y . The angle is expressed in degrees, between 0 and 360. The quadrant of the angle is determined by the signs of the inputs.

COS Outputs the cosine of its input (as an angle in degrees).

INT Example:

```
PRINT INT 5.6
5
PRINT INT -5.6
-5
```

Takes one numeric input and converts it to an integer by truncating any fractional part.

PRODUCT Example:

```
PRINT (PRODUCT 3 4 5)
60
```

Takes a variable number of inputs (default is 2) and outputs their product.

QUOTIENT Example:

```
PRINT QUOTIENT 5 2
2
```

Outputs the integer quotient of its first input divided by its second. If the inputs are not integers, it first truncates them.

RANDOM Example:

```
PRINT RANDOM 100
53
PRINT RANDOM 100
27
```

Takes one input, a positive integer n , and outputs an integer between 0 and $n - 1$.

REMAINDER Example:

```
PRINT REMAINDER 13 10
3
```

Outputs the integer remainder of its first input modulo its second. If the inputs are not integers, it first truncates them.

RERANDOM Takes no inputs. Re-initializes the seed for the random number generator.

ROUND Example:

```
PRINT ROUND 5.6
6
PRINT ROUND -5.6
-6
```

Outputs the nearest integer to its input.

SIN Outputs the sine of its input (as an angle in degrees).

SQRT Takes a positive number as input and outputs the square root of that number.

SUM Example:

```
PRINT (SUM 1 2 3 4)
10
```

Takes a variable number of inputs (default is 2) and outputs their sum.

10.3. Word and List Operations

In addition to numbers, Logo also includes operations for dealing with words (strings of characters) and lists (structured collections of data.)

BUTFIRST Abbreviated BF
Example:

```
PRINT BUTFIRST [THIS IS A LIST]
IS A LIST
PRINT BUTFIRST "ABRACADABRA
BRACADABRA
```

If input is a list, outputs a list containing all but the first element. If input is a word, outputs a word containing all but the first character. Gives an error when called with the empty word or the empty list as input.

BUTLAST Abbreviated BL
Example:

```
PRINT BUTLAST [THIS IS A LIST]
THIS IS A
PRINT BUTLAST "ABRACADABRA
ABRACADABR
```

If input is a list, outputs a list containing all but the last element. If input is a word, outputs a word containing all but the last character. Signals an error when called with the empty word or the empty list as input.

COUNT Example:

```
PRINT COUNT [FEE FIE FOE FUM]
4
```

Takes a list as input and outputs the number of items in the list.

FIRST Example:

```
PRINT FIRST [THIS IS A LIST]
THIS
PRINT FIRST "ABRACADABRA
A
```

If input is a list, outputs the first element. If input is a word, outputs the first character. Signals an error when called with the empty word or the empty list as input.

FPUT Example:

```
PRINT FPUT [A B] [C D]
[A B] C D
```

The second input must be a list. Outputs a list consisting of the first input followed by the elements of the second input.

ITEM Example:

```
PRINT ITEM 3 [FEE FIE FOE FUM]
FOE
PRINT ITEM 2 [[HOBBY HORSE] [PACK RAT] BANANA]
PACK RAT
```

Takes a number n and a list as input, and outputs the n th item in the list. Signals an error if n is greater than the COUNT of the list.

LAST Example:

```
PRINT LAST [THIS IS A LIST]
LIST
PRINT LAST "ABRACADABRAX
X
```

If input is a list outputs the last element. If input is a word, outputs the last character. Signals an error when called with the empty word or the empty list as input.

LIST Example:

```
PRINT LIST "A "B
A B
PRINT (LIST "A "B [1 2 3] "C)
A B [1 2 3] C
```

Takes a variable number of inputs (two by default) and outputs a list of the inputs.

LPUT Example:

```
PRINT LPUT "Z [W X Y]
W X Y Z
PRINT LPUT [A B] [C D]
C D [A B]
```

Second input must be a list. Outputs a list consisting of the elements of the second input followed by the first input.

SENTENCE Abbreviated SE

Example:

```
PRINT SENTENCE "HELLO "THERE
HELLO THERE
PRINT SENTENCE [THIS IS] [A LIST]
THIS IS A LIST
PRINT (SENTENCE "THIS [IS] [A LIST])
THIS IS A LIST
PRINT SENTENCE [[HERE IS] A] [NESTED LIST]
[HERE IS] A NESTED LIST
```

Takes a variable number of inputs. (The default is two). If inputs are all lists, combines all their elements into a single list. If any inputs are words, they are regarded as one-word lists in performing this operation.

WORD Example:

```
PRINT WORD "MISH "MASH
MISHMASH
PRINT (WORD 123 45 678)
12345678
```

Variable number of inputs (default is two). Outputs a word that is the concatenation of the characters of its inputs (which must be words).

10.4. Defining and Editing Procedures

TO and EDIT are the most commonly used operations for creating and changing procedures. But Logo includes some other operations that allow more advanced manipulation of procedure definitions.

DEFINE Example:

```
DEFINE "PTSUM [[X Y] [PRINT :X] [PRINT :X + :Y]]
defines the procedure
```

```

TO PTSUM :X :Y
PRINT :X
PRINT :X + :Y
END

```

Takes two inputs. First is a name, and second is a list whose elements are a list of inputs and a list for each line, and defines a procedure accordingly. Note that you normally use TO rather than DEFINE in order to define procedures. DEFINE is useful for writing procedures that define other procedures, as in the extended INSTANT system described in section 9.2.1.

EDIT Abbreviated ED

Example:

```

EDIT "SQUARE
{sets up procedure SQUARE for editing}

```

Enters the procedure editor with a given procedure. Can also take as input a list of procedure names to be edited. If no input is specified, enters with the previous contents of the screen-editor buffer, or a blank screen if the previous contents are unretrievable.

EDNS With no inputs, enters the definitions of all variables and their values into the edit buffer, so these can be edited. Can also take a package name or a list of packages as input, in which case it enters only the information in the designated packages.

END Terminates a procedure definition that is typed into the editor. It is not necessary to type END at the end of the final definition, but if you are defining more than one procedure at a time, the separate procedure definitions must be separated by END statements.

TEXT Example:

```

TO PTSUM :X :Y
PRINT :X
PRINT :X + :Y
END

PRINT TEXT "PTSUM
[X Y][PRINT :X][PRINT :X + :Y]

```

Takes a procedure name as input and outputs procedure text as a list, whose format is as described under DEFINE.

TO Begins procedure definition. Enters definition mode if typed as a direct command.

10.5. Conditional Expressions

Logo includes two basic facilities for allowing the user to write programs that perform tests and do different things depending on the outcomes. One is the IF construct that is common to many computer languages. The other, TEST . . . IFTRUE . . . IFFALSE, is less common but often simpler to use.

AND Example:

```
PRINT (AND (1 + 1 = 2) (5 = 4) (1 = 1))
FALSE
```

Takes a variable number of inputs (default is two). Each input should be either TRUE or FALSE. Outputs TRUE if all are TRUE, otherwise outputs FALSE.

OR Example:

```
PRINT (OR (1 + 1 = 2) (5 = 4) (1 = 1))
TRUE
```

Takes variable number of inputs (default is two) and outputs TRUE if at least one is TRUE, otherwise outputs FALSE.

IF Example:

```
IF :X = 5 [STOP] [PRINT "HELLO]
```

Logo's basic condition form is IF {condition} [{action1}] [{action2}]. The {condition} is tested. If it is true {action1} is performed. If it is false, {action2} is performed. The {action2} part need not be present.

IFFALSE Abbreviated IFF

Executes a list of commands only if result of preceding TEST was false. See TEST.

IFTRUE Abbreviated IFT

Executes a list of commands only if result of preceding TEST was true. See TEST.

NOT Example:

```
IF NOT (1 = 1) [PRINT "HELP]
HELP
```

Outputs TRUE if its input is FALSE, FALSE if its input is TRUE.

TEST Example:

```
TEST 12 = WORD 1 2
IFFALSE [PRINT "NO]
IFTRUE [PRINT "YES]
YES
```

Tests a condition to be used in conjunction with IFTRUE and IFFALSE.

10.6. Predicates Used with Conditional Expressions

The conditional expressions of the previous section make use of predicates. A predicate can be any procedure that outputs the word TRUE or the word FALSE. Here are the predicates that are built into Logo.

> Example:

```
IF :X > :Y [STOP]
```

Outputs TRUE if its first input is greater than its second, FALSE otherwise.

< Outputs TRUE if its first input is less than its second, FALSE otherwise

= Example:

```
PRINT 20 = WORD 2 0
```

TRUE

```
PRINT "A = [A]
```

FALSE

```
PRINT [A B] = SENTENCE "A "B
```

TRUE

If both inputs are numbers, compares them to see if they are numerically equal. If both inputs are words, compares them to see if they are identical character strings. If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs TRUE or FALSE accordingly.

EMPTY? Example:

```
PRINT EMPTY? BUTFIRST [LOLLIPOP]
```

TRUE

```
PRINT EMPTY? BUTFIRST "X
```

TRUE

Takes one input. Outputs TRUE if the input is the empty word or the empty list, FALSE otherwise.

EQUALP Example:

```
PRINT EQUALP "A [A]
```

FALSE

Takes two inputs. This is a prefix form of =.

LISTP Outputs TRUE if its input is a list, FALSE otherwise.

MEMBERP Example:

```
PRINT MEMBERP "A [ONCE OPON A TIME]
TRUE
```

Takes an item and a list as input. Outputs TRUE if the item is a member of the list, FALSE otherwise.

NUMBERP Outputs TRUE if its input is a number, FALSE otherwise.

WORDP Outputs TRUE if its input is a word, FALSE otherwise.

10.7. Controlling Procedure Execution

OUTPUT Abbreviated OP

Takes one input. Causes the current procedure to stop and output the result to the calling procedure.

REPEAT Example:

```
REPEAT 3 [PRINT "HELLO]
HELLO
HELLO
HELLO
```

Takes a number and a list as input. RUNs the list the designated number of times.

RUN Example:

```
MAKE "X [PRINT]
RUN SENTENCE :X 5
5
```

Takes a list as input. Executes the list as if it were a typed in command line.

STOP Causes the current procedure to stop and return control to the calling procedure.

10.8. Input and Output

ASCII Takes a character as input and outputs the number that is the ASCII code of that character.

BUTTONP Takes a number 0 through 2 as input and outputs TRUE or FALSE depending on whether the button on the corresponding paddle is pressed.

CHAR Takes an integer as input and outputs the character whose ASCII code is that integer.

CLEARTEXT Takes no inputs. Clears the text screen and moves the cursor to the left edge of the screen on the first available text line.

- CURSOR** Takes no inputs. Outputs a list of two integers that specifies the position of the cursor. See SETCURSOR.
- KEYP** Takes no inputs. Outputs TRUE if a keyboard character is pending (i.e., the character input buffer is not empty), otherwise outputs FALSE.
- PADDLE** Takes a number 0 through 3 as input and outputs a number 0–255 depending on the setting of the appropriate paddle dial.
- PRINT** Abbreviated PR
Example:
PRINT "HI
HI
(PRINT "HELLO "OUT "THERE)
HELLO OUT THERE
PRINT [HELLO OUT THERE]
HELLO OUT THERE
- Takes a variable number of inputs (default is 1). Prints them on the screen, separated by spaces, and moves cursor to the next line. When PRINT prints lists, the outermost pair of brackets is not printed.
- .PRINTER** Takes as input a number *n* designating a slot on the Apple card. After executing this command, all further typeout is directed to the device plugged in to slot *n* rather than to the screen. .PRINTER 0 restores output to the screen. If *n* is between 9 and 15, then output is sent both to the screen and to the device in slot *n* – 8.
- READCHAR** Abbreviated RC
Takes no inputs. Outputs the least recent character in the character buffer, or if empty, waits for an input character.
- READLIST** Abbreviated RL
Takes no inputs. Waits for an input line to be typed, terminated with RETURN. Outputs the line (as a list).
- SETCURSOR** Takes as input a list of two positive integers, which are interpreted as a column and a row and positions the input cursor there. Columns are 0–39, rows are 0–23. (0,0) is upper left.
- SHOW** Like PRINT, except that it prints the outer brackets around lists. For example:

```

PRINT [HI]
HI
SHOW [HI]
[HI]
PRINT "HI
HI
SHOW "HI
HI

```

SHOW is often useful in debugging programs that deal with lists.

TYPE Like PRINT, but does not move cursor to the next line after printing. With multiple inputs, does not print spaces between the inputs.

WAIT Takes one input, an integer *n*. Causes Logo to wait for *n* 60ths of a second.

10.9. Naming

DEFINEDP Outputs TRUE if its input is the name of a procedure, FALSE otherwise.

LOCAL Normally, variable names that are declared as inputs are *local* to the procedure in which they are declared, while other variable names are not. (See section 5.5.1 for an explanation of local and global names.) The LOCAL command allows you to declare other names besides inputs to be local to a given procedure. For example:

```

TO DIST :X :Y
LOCAL "D
MAKE "D (:X *:X) + (:Y *:Y)
OUTPUT SQRT :D
END

```

is a procedure that uses D as a temporary variable. By declaring D to be local, we insure that this will not conflict with any other variable called D outside of the procedure DIST.

MAKE Example:

```

MAKE "APPLE 50
PRINT :APPLE
50

```

Takes two inputs, the first of which must be a word. Assigns the second input to be the value associated with the first input.

NAME This is equivalent to MAKE with the order of the inputs reversed. Example:

```
NAME [BATMAN SUPERMAN BOGEYMAN] "SUPERHEROES
PRINT :SUPERHEROES
BATMAN SUPERMAN BOGEYMAN
```

NAMEP Outputs TRUE if its input has a value associated with it.

THING Example:

```
MAKE "APPLE 50
PRINT THING "APPLE
50
```

Outputs the value of its input (which must be a word).
THING "XXX is normally abbreviated as :XXX.

10.10. Filing and Managing Workspace

Workspace consists of all currently defined procedures and all names and their associated values. Workspaces can be stored in files on disk.

BURY Takes a package name as input, and declares all information in that package to be "buried," so that it will not be affected by POALL or ERALL, and will not be SAVED onto disk. See section 4.1.3 for more information.

CATALOG Takes no inputs. Prints the names of files contained on the currently mounted disk.

ERALL With no inputs, erases all (nonburied) information from workspace. Can also take a package name or list of packages as input, in which case it erases only the information in the specified packages.

ERASE Abbreviated ER
Example:

```
ERASE "SQUARE
{gets rid of the procedure SQUARE}
```

Erases designated procedure from workspace. Can also take as input a list of procedures to be erased.

ERASEFILE Example:

```
ERASEFILE "MYFILE
{gets rid of the file whose name is MYFILE}
```

Removes a file from the disk. Takes file name as input.

ERN Example:

```
MAKE "APPLE 5
ERNAME "APPLE
PRINT :APPLE
APPLE HAS NO VALUE
```

Takes a name as input and removes that name from the library. Can also take as input a list of names to be erased.

ERNS With no input, erases all (unburied) names from workspace. Can also take as input a package name or a list of packages in which case only the names in the designated packages are erased.

ERPS With no input, erases all (unburied) procedures from workspace. Can also take as input a package name or a list of packages, in which case only the procedures in the designated packages are erased.

LOAD Example:

```
LOAD "MYFILE
```

Loads a file from disk. Destroys any graphics display. Takes file name as input. Can also take a package name as an optional second input, in which case the information in the file is loaded into the designated package.

PACKAGE Takes as inputs a package name and a symbol, and declares that symbol to be part of the specified package. Example:

```
PACKAGE "DRAWINGS "MONALISA
```

Declares MONALISA to be part of the DRAWINGS package. See section 4.1.3 for more details. The second input can also be a list of names to be packaged.

PKGALL Takes a package name as input and declares all currently unpackaged names to be part of the designated package.

PO If given a procedure name as input, prints out the text of the procedure. Can also take a list of procedure names as input.

POALL With no input, prints all (unburied) procedure definitions and values of all variables. Can also take as input a package name or a list of package names, in which case only the information in the designated packages are printed.

PONS With no input, prints the values of all (unburied) variables. Can also take as input a package name or a list of package names, in which case only the information in the designated packages are printed.

- POPS** With no input, prints the definitions of all (unburied) procedures. Can also take as input a package name or a list of package names, in which case only the information in the designated packages are printed.
- POTS** With no input, prints the titles of all (unburied) procedures. Can also take as input a package name or a list of package names, in which case only the information in the designated packages are printed.
- SAVE** Example:
 SAVE "MYFILE
 Saves the contents of the workspace on disk. Destroys any graphics display. Takes file name as input. Buried information will not be saved. Can also take a package name or a list of packages as an optional second input, in which case only the information in the designated packages is saved.
- UNBURY** Takes a package name as input and "unburies" the information in that package. See BURY and section 4.1.3 for more information.

10.11. Debugging Aids

- CONTINUE** Abbreviated CO
 Resumes execution after a PAUSE. Can take an input, in which case it returns this as its value. This is useful if you have paused (by typing CTRL-Z) during a READLINE, because it allows you to use the input to CONTINUE to specify the value to be returned as the value of the READLINE.
- PAUSE** Takes no inputs. Stops execution and allows command lines to be evaluated in the current local environment. Equivalent to interrupt character CTRL-Z. Execution can be resumed with CONTINUE, provided no errors have occurred during the pause. The command THROW "TOPLEVEL can be used to abort the pause the return to command level.

10.12. Editing Commands

This section describes the commands that are used with the procedure editor. Most of these editing commands are also available to correct typing errors in command lines. Notice that a "control character" is typed by holding down the control key and typing the character. For example, to type CTRL-V, hold down the key marked CTRL and press V. On the other hand, an "escape character" is typed by pressing ESC and then the key. That is, to type ESC-V first type ESC and then type V.

left arrow key ←

Rubs out the character immediately to the left of the cursor and moves the cursor one space to the left.

right arrow key →

Moves the cursor one character to their right *without* rubbing out any character.

REPT key If you type a character and hold down this key, the keyboard repeatedly transmits that character for as long as REPT is held down. This is useful in conjunction with some of the other editing keys.

CTRL-A Moves the cursor to the beginning of the current line.

CTRL-B Moves the cursor one position to the left, without rubbing out any character.

CTRL-C Exits the editor. Processes the edited text.

CTRL-D Deletes the character at the current cursor position, that is, the character over which the cursor is flashing.

CTRL-E Moves the cursor to the end of the current line.

CTRL-F Same as →

CTRL-G In edit mode, exits the editor without processing the edited text. Otherwise, stops execution and returns control to top level.

CTRL-H Same as ←

CTRL-K Deletes all characters on the current line to the right of the cursor. If typed while the cursor is at the right end of the line, merges the current line with the next line.

CTRL-L In edit mode, scrolls the text so that the line containing the cursor is approximately in the center of the screen.

CTRL-M Equivalent to pressing RETURN.

CTRL-N Moves the cursor down one line.

CTRL-O Opens the new line at the cursor position. That is, typing CTRL-O is equivalent to typing RETURN and restoring the cursor to its original position.

- CTRL-P In edit mode, moves the cursor up one line.
- CTRL-U Same as →
- CTRL-V When editing more than one screenful of text, moves the cursor approximately one screenful of text forward, or to the end of the buffer if not that much text follows the cursor.
- CTRL-Y Inserts at the current cursor position a copy of the text that was previously killed with CTRL-K. (Therefore, to move a line from one place to another, you can kill it, move the cursor to a new position, and “yank” the line back with CTRL-Y.) In draw or nodraw mode, retrieves previous input line so that it can be edited and/or reexecuted.
- ESC-V When editing more than one screenful of text, moves the cursor approximately one screenful of text backwards, or to the beginning of the buffer if not that much text precedes the cursor.
- ESC-< Moves the cursor to the beginning of the editor buffer.
- ESC-> Moves the cursor to the end of the editor buffer.

10.13. Other Control Characters

This section describes control characters used in Logo other than for editing. Control-character operations are different from Logo primitives in that they can be typed while a program is running and take effect as soon as they are typed.

- CTRL-G In edit mode, exits the editor without processing the edited text. In draw or nodraw mode, stops execution and returns control to top level.
- CTRL-L In graphics mode, gives full graphics screen.
- CTRL-Q This is used to insert “funny characters” in words. For example, the bracket character [cannot normally be part of a word. However, you can include it in a word by preceding it with a CTRL-Q when you type it in. CTRL-Q echoes on the screen as a backslash (\). Example:


```
MAKE "FUNNY "AB\[CD
PRINT :FUNNY
AB/CD
```
- CTRL-S In graphics mode, gives mixed text/graphics screen.
- CTRL-T In graphics mode, gives full text screen.

- CTRL-W Stops program execution. Repeatedly typing CTRL-W causes Logo to stop after printing the next line (or the next list element if lists are being printed). Typing any other character will resume normal processing.
- CTRL-Z Causes Logo to PAUSE.

10.14. Primitives for Handling Properties

Properties are a built-in feature that Logo provides for arranging information in associative tables. See section 9.1.2.

- GPROP Takes as inputs a symbol and a property name. Returns the value of the associated property for that symbol, or the empty list if there is no such property.
- PLIST Takes a symbol as input and returns that symbol's properties, in list form.
- PPROP Takes a symbol, a property, and a value as inputs. Associates that property value to the symbol. Example:
- ```
PPROP "BASIC "POWER "LIMITED
PPROP "BASIC "SIZE "SMALL
GPROP "BASIC "POWER
LIMITED
PRINT PLIST "BASIC
SIZE SMALL POWER LIMITED
```
- PPS With no input, prints the properties of everything in workspace. Can also take a package name or a list of packages as input, in which case it prints the properties of the information in the designated packages.
- REMPROP Takes a symbol and a property name as inputs and removes the corresponding property from the symbol.

#### 10.15. Miscellaneous and Advanced Commands

The commands in this section, for the most part, are concerned with advanced features of the Logo system that have not been discussed in this book, and with aspects of the language that are highly specific to the Apple Logo implementation. Consult the reference manual for more details than are given here.

- .BPT Takes no inputs. Break out of Logo into the Apple monitor. To reenter Logo, type 803G followed by RETURN.
- CATCH This command, together with its companion THROW, provides a means to "abort" procedure executions. CATCH takes two inputs: a name and a list of instructions to to run. It runs the



list, but, if at any point while the list is being run, Logo comes across a **THROW** with the same name as the **CATCH**, then execution will return immediately to the statement following the **CATCH**. In particular, since running the instruction list will in general call procedures which call other procedures, and so on, executing the **THROW** will return through as many levels of procedure calls as necessary in order to return to the **CATCH**. **CATCH** and **THROW** thus provide a mechanism for "non-local transfer of control," which should be used with caution since it can lead to obscure programs. One very important use for this mechanism is to allow the user to control what Logo does when it runs across an error. Logo actually executes a **THROW "ERROR** when it hits any error, so **CATCH "ERROR** can be used to intercept errors without having the program stop. The **ERROR** primitive provided in Logo can then be used to retrieve information about the error so that the program can take appropriate action. See the reference manual for these and other uses of **CATCH** and **THROW**.

- .CONTENTS** Takes no inputs. Outputs a list of all symbols that are currently being used (variable names, procedure names, primitives, properties, packages, etc.)
- COPYDEF** Takes two inputs, a symbol **A** and a symbol **B**, and causes **A** to have the same definition as **B**.
- .DEPOSIT** Takes two numeric inputs, an address and a value, and deposits a byte of data at a designated memory location.
- DISK** Takes no inputs. Outputs a list of three numbers: the disk drive number, the slot number of the disk drive, and the volume number of the disk most recently used with **CATALOG** or **SETDISK**.
- ERROR** Outputs information about the most recent error. This can be used together with **CATCH** and **THROW** to change the way the system behaves in response to various errors. See the reference manual for details.
- .EXAMINE** Outputs the value of the byte at the specified address.
- GO** Example:
 

```

 TO TRIANGLE :STRING
 IF :STRING = " THEN STOP
 LABEL "LOOP
 PRINT :STRING
 MAKE "STRING BUTFIRST :STRING
 GO "LOOP

```

Compare this example with the TRIANGLE procedure of section 5.3. GO takes a word as input and transfers to the line with that label. You can only GO to a label within the same procedure. Labels are defined by the LABEL command.<sup>1</sup>

- LABEL** See GO.
- NODES** Outputs the number of currently free nodes. This is a measure of how much storage is available in the workspace.
- PRIMITIVEP** Outputs TRUE if its input is the name of a Logo primitive, FALSE otherwise.
- RECYCLE** Forces a garbage collection, reclaiming unused storage.
- REPARSE** Causes reparsing of Logo procedures. When Logo runs across a procedure whose definition has been changed, it reparses it as necessary, so that the user is normally unaware that parsing is going on. On the other hand, this reparsing operation takes time and can cause unexpected waits while Logo is running. The REPARSE primitive allows you to control when this reparsing happens.
- SCRUNCH** Takes no inputs. Returns the current setting of the graphics screen aspect ratio. See SETSCRUNCH.
- SETSCRUNCH** Changes the vertical scale at which Logo graphics are drawn. Takes one numeric input and uses this to change the scale factor. The default value for the factor is 0.8. This command is included because not all TV monitors have the same amount of vertical deflection. Consequently, turtle programs that are supposed to draw squares and circles may instead appear to draw rectangles and ellipses. If so, the SETSCRUNCH command can be used to attempt to compensate for the distortion. Changing the factor changes the limits for permissible y coordinates.<sup>2</sup>

---

<sup>1</sup> GO is occasionally useful but is easily abused and can lead to obscure programs. In Logo, you can almost always avoid the need to use GO by taking advantage of REPEAT and/or procedure calls. Iteration constructs (WHILE, FOR, and so on) can also be implemented by using RUN, as illustrated in Section 9.2.4. As the TRIANGLE procedure above shows, one can in fact use Logo to program in a style that is typical of most BASIC programs. That would be like pouring ketchup over caviar.

---

<sup>2</sup> There is a problem with using values of SETSCRUNCH that are too different from 0.8. Although lines will be drawn at the correct angle, the turtle pointer may not always appear to be pointing exactly along the line.

- SETDISK** Can take one, two, or three inputs, which specify a disk drive number, a slot number for the drive, and a volume number for the disk. Telling Logo to verify the volume number can be useful, since it can protect you from accidentally using the wrong diskette.
- THROW** An advanced feature providing “non-local” returns from procedures. See the brief description under **CATCH** and refer to the reference manual for details and examples.

## 10.16. Error Messages

When Logo encounters an error, it signals that fact by halting program execution and printing a message of the form:

```
{message} IN {procedure}:
{line}
```

For example:

- *I DON'T KNOW HOW TO FORWAXD IN BOX:*  
*FORWAXD 100*

In general, {message} is a description of the error, {line} is the line at which the error occurred, {procedure} is the name of the procedure containing that line. If the error is caused by the “last thing the procedure does,” then Logo will print

```
{message}:
JUST BEFORE LEAVING {procedure}
```

instead of printing the actual last line of the procedure. This section lists the most common error messages generated by Apple Logo and describes the conditions that commonly cause them. See the reference manual for a complete list of error messages.

- *I DON'T KNOW HOW TO {something}*  
This happens when Logo does not recognize the name of the procedure you are trying to run. Common causes are that you forgot to define the procedure in question, or that you used the wrong name. Typing errors also commonly cause this. For example, if you type *FORWAXD 100* instead of *FORWARD 100*, you will get the error *I DON'T KNOW HOW TO FORWAXD*.
- *{something} HAS NO VALUE*  
This happens when you refer to the value of a name, but there is no such name in the environment. The causes are similar to those for the “no procedure” error message. Another cause is confusion between the *local* variables in a procedure and the global variables. For example, defining and running the procedure

```

TO INC :X
OUTPUT :X + 1
END

```

creates a variable X that is local to INC, but this does not mean that there is a global variable named X.

- *NOT ENOUGH INPUTS TO {procedure}*  
A procedure was called with too few inputs.
- *{primitive} DOESN'T LIKE {data} AS INPUT*  
This happens when you try to use an operation with a kind of data that it cannot handle. For example:  

```
PRINT 1 + "X
```

  
results in *+ DOESN'T LIKE X AS INPUT.*

- *I DON'T KNOW WHAT TO DO WITH {data}*  
This occurs in procedure when you generate some data and then don't say what to do with it. (In most cases, you probably meant to OUTPUT it.) For example:

```

TO SQUARE :X
:X * :X
END

```

```

SQUARE 5
I DON'T KNOW WHAT TO DO WITH 25:
JUST BEFORE LEAVING SQUARE

```

The problem here is that SQUARE should have an OUTPUT at the beginning of its first line.

- *{procedure} DIDN'T OUTPUT TO {something}*  
This happens when you try to use the value returned by a procedure, but the procedure didn't output anything. For example,

```

TO PRINT.SQUARE :X
PRINT :X * :X
END

```

```

FORWARD PRINTSQUARE 4
16
PRINTSQUARE DIDN'T OUTPUT TO FORWARD

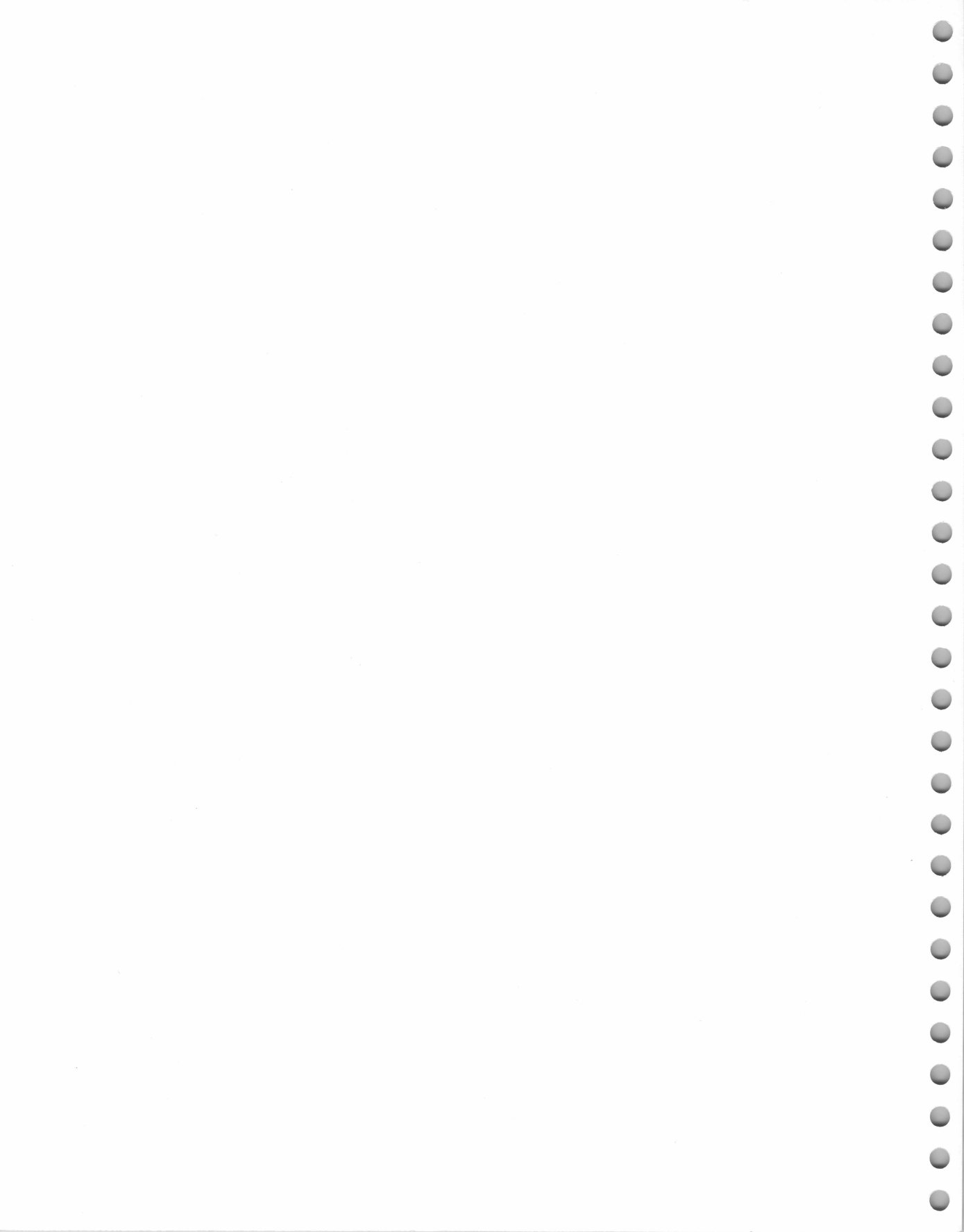
```

- *OUT OF SPACE*  
This message indicates that you have used up all available storage.

- **{procedure} IS ALREADY DEFINED**  
This happens when you try to define a procedure that is already defined. For example, if you type **TO SQUARE** when you already have a procedure **SQUARE**, Logo will complain. (This error can be avoided by always using **EDIT** to define procedures.)
- **NUMBER TOO BIG**  
This happens when an arithmetic operation generates a number too large or too small for Logo to handle.
- **CAN'T DIVIDE BY ZERO**  
This happens when the **QUOTIENT**, **REMAINDER**, or **/** operation is called with zero as the divisor.
- **TURTLE OUT OF BOUNDS**  
This happens when Logo is in **FENCE** mode and you try to move the turtle beyond the screen boundary.

## Appendix

---



## APPENDIX

## The TI LOGO<sup>(TM)</sup> Implementation of Logo

---

The Logo implementation described in this book was developed by Logo Computer Systems Incorporated and is based on prototype versions of Logo developed over a period of years at MIT. In this appendix we compare it with a version of Logo developed at MIT for the TI 99/4 and 99/4A personal computers. The TI implementation was developed with support by Texas Instruments, Incorporated. Although both versions of Logo are based on the same fundamental ideas, there are significant differences between the two implementations, primarily due to different hardware capabilities.

The most significant difference has to do with the graphics capabilities of the two computers. The Texas Instruments computers allow for dynamic animation graphics, implemented in the form of "sprites," turtle-like objects that can take on different shapes and colors and move freely around the screen. There can be many different sprites of different shapes and colors moving on the screen at any given time. Sprites make possible an entire range of exciting applications that are not presently possible with implementations of Logo for the Apple II.

Another graphics feature of the TI computers that is not shared by the Apple II is the ability to create "tiles," that is, user-defined character shapes that can be printed anywhere on the screen. This makes it possible to create certain kinds of screen images that cannot be easily drawn by the turtle. It also allows a user to define a new set of alphanumeric characters if desired.

A corresponding limitation of the TI graphics is that turtle drawings are made by drawing lines on a series of blank character tiles. When the available supply of character tiles is exhausted, the turtle must stop drawing. This means that turtle drawings can occupy only a limited portion of the TV screen area.

The graphics capabilities of TI LOGO are described in the technical reference manual that accompanies it and will not be covered further here. Other major differences between TI and Apple implementations of Logo should also be mentioned. The TI implementation of Logo is limited to integer arithmetic.



Projects that require decimal arithmetic or transcendental functions cannot easily be carried out. There are also some differences in the way that the TI version of Logo deals with words and lists. These will be described in appropriate sections of this appendix.

Despite differences, the TI and Apple versions of Logo are fundamentally very similar. In the rest of this appendix, we will describe differences between Apple Logo and TI LOGO to allow TI users to carry out most of the activities and projects described in this book. This appendix was prepared with the assistance of Dan Watt.

## CHAPTER 1

To enter turtle drawing mode use the command TELL TURTLE. The NOTURTLE command is used to leave the mode.

The keyboard commands used for editing or interrupting the computer are different than those used for the Apple. Since the TI 99/4 and 99/4A computers also have very different keyboards, keyboard commands will not be discussed here. Refer to TI LOGO documentation for information about use of the keyboard. TO and EDIT have the same effect in TI LOGO. They both cause Logo to enter the editor, as described in section 1.3.2. Procedure names should not be quoted when using EDIT and PO. There is no procedure definition mode, as described in section 1.3.1. Procedure editing is very similar except for the different keys involved in the editing commands. To print out a list of procedure titles, use PP (printout procedures) instead of POTS. To print out all procedures use PA (printout all) instead of POALL.

TI LOGO allows sixteen different colors, which have names as well as numbers. The turtle's pencolor is set by the SETCOLOR command (abbreviated SC). Using a color name, SETCOLOR :WHITE, has the same effect as using a number, SETCOLOR 15. The background color is changed using the COLORBACKGROUND command (abbreviated CB) with a number or color name as input. To set the turtle's position, use SXY, SX, SY and SH instead of SETPOS, SETX, SETY and SETH. SXY takes two numbers as inputs, SXY 10 10, rather than a list as used by SETPOS, SETPOS [ 10 10 ].

## CHAPTER 2

There is an important difference in the way the TI editor works when defining procedures with inputs. The TI LOGO editor only accepts a one-word procedure name when entering the editor. To define a procedure with inputs such as TO SQUARE :SIDE (section 2.1.), first type TO SQUARE or EDIT SQUARE and press the ENTER key. Then type in the rest of the line, :SIDE. If you type the entire title line, TO SQUARE :SIDE, before entering the editor, only the first word after TO will be stored, and you will have to retype

:SIDE anyway.

The construction of circle and arc procedures (section 2.1.3) is different because TI LOGO is limited to integer arithmetic. If you use  $22/7$  as an approximation to  $\pi$ , the result will be 3, the integer quotient. This allows much less flexibility in relating the forward step used by the turtle in drawing a circle to the radius of the circle. The following approximate circle and arc procedures have been acceptable for many applications:

```
TO RCIRCLE :RADIUS
REPEAT 12 [RCP :RADIUS]
END
```

```
TO RARC :RADIUS
REPEAT 3 [RCP :RADIUS]
END
```

RCP :RADIUS draws one "circle piece" curving to the right that corresponds to one-twelfth of a circle. RARC therefore draws an arc equivalent to a quarter circle:

```
TO RCP :RADIUS
RIGHT 15
FORWARD :RADIUS / 2
RIGHT 15
END
```

The size of the forward step is determined by using the formula  $2 \times \pi \times R = 12 \times \text{size}$ , for a twelve-sided "circle" with the integer approximation  $\pi = 3$ . It is difficult to write a general procedure to draw "circles" with more than twelve sides.

When using the IF command (section 2.2.2), do not enclose the action in square brackets:

```
IF :NUMBER = 0 [STOP] (Apple Logo)
IF :NUMBER = 0 STOP (TI LOGO)
```

## CHAPTER 3

All of the project ideas given in Chapter 3 can be carried out in TI LOGO by using approximate circle procedures similar to the ones given above instead of the procedures given on page 45. Some turtle drawings may terminate before being completed because of the limitation on the total area that can be covered by turtle drawings.

## CHAPTER 4

PO, ERASE and EDIT (section 4.1) can take only single procedure names as inputs. Procedure names should not be quoted. EDIT without any input causes a shift to a blank edit buffer. To printout a listing of procedure titles, use the

command PP ("printout procedures") instead of POTS. To printout the entire workspace, use PA ("printout all") instead of POALL. To printout just the names, use PN instead of PONS. There is no equivalent for POPS.

The filing system (section 4.2) is totally different and will not be discussed here. Refer to TI LOGO reference documents for filing and hard copy information. A command called TRACEBACK will provide information about which procedures are currently active if it is called during a pause (FCTN 7 for the TI 99/4A and SHIFT A for the TI 99/4).

There is no equivalent in TI LOGO to the PACKAGE capability of Apple Logo, as described in section 4.1.3, nor the ability to BURY parts of the workspace.

## CHAPTER 5

Numbers, words, and lists are handled somewhat differently in TI Logo than in Apple Logo. Numbers are *not* treated as words, so that word-related commands such as WORD, FIRST, BUTFIRST, LAST, and BUTLAST do not accept numbers as inputs. On the other hand it is possible to have a word all of whose characters are digits:

```
PRINT 25
25
PRINT "25
25
```

However, the *word* "25 is *not* the same as the *number* 25. For example, arithmetic commands will not accept "25 as input. This can be confusing because the error messages that result from using inappropriate inputs do not distinguish between numbers and words:

```
PRINT FIRST "25
2
PRINT FIRST 25
FIRST DOESN'T LIKE 25 AS INPUT
PRINT "25 + 5
+ DOESN'T LIKE 25 AS INPUT
PRINT 25 + 5
30
```

Decimal fractions are not allowed by TI LOGO, and only integer arithmetic is provided. Quotients are always truncated to the next lower integer:

```
PRINT 5 / 2
2
```

TI LOGO does not include a REMAINDER command (section 5.1.2). You will need the following procedure whenever REMAINDER is used in this book:

```
TO REMAINDER :NUMBER :DIVISOR
 OUTPUT :NUMBER - (:NUMBER / :DIVISOR) * :DIVISOR
END
```

```
PRINT REMAINDER 5 3
2
```

RANDOM takes no input and always outputs a number from zero to nine. To simulate the behavior of the RANDOM command in Apple Logo you will need to create a procedure, RANDOM1 :NUMBER, that outputs a random number between zero and :NUMBER - 1. This requires a complicated set of subprocedures. First, we need a procedure, RANDOMNUMBER :DIGITS, which outputs a random number with the number of digits specified as its input. RANDOMNUMBER 2 outputs a two-digit random number, etc. (Numbers larger than 32,767 are meaningless in TI LOGO since they exceed the limits of its integer arithmetic.) Using RANDOMNUMBER, RANDOM1 becomes:

```
TO RANDOM1 :NUMBER
 MAKE "RAND1 RANDOMNUMBER DIGITS :NUMBER
 IF :RAND1 < :NUMBER THEN OUTPUT :RAND1
 OUTPUT RANDOM1 :NUMBER
END
```

DIGITS outputs the number of digits in a number:

```
TO DIGITS :NUMBER
 IF :NUMBER / 10 = 0 THEN OUTPUT 1
 OUTPUT 1 + DIGITS (:NUMBER / 10)
END
```

RANDOMNUMBER requires the subprocedure RANDOMDIGIT :DIGITS that outputs a random digit multiplied by the correct power of ten. RANDOMDIGIT uses the TI LOGO command, RANDOM.

```
TO RANDOMNUMBER :DIGITS
 IF :DIGITS = 0 THEN OUTPUT 0
 OUTPUT (RANDOMNUMBER :DIGITS - 1) +
 (RANDOMDIGIT :DIGITS)
END
```

Type as one line. See page 9.→

```

TO RANDOMDIGIT :DIGITS
OUTPUT (RANDOM * TENPOWER :DIGITS)
END

```

```

TO TENPOWER :DIGITS
IF :DIGITS = 0 THEN OUTPUT 1
OUTPUT 10 * (TENPOWER :DIGITS - 1)
END

```

Use RANDOM1 with the above subprocedures in place of RANDOM as used in projects throughout this book. You can either copy them as a set of “black boxes” or trace through them carefully to see how they work. There is no equivalent to the RERANDOM command used Apple Logo.

### Section 5.3. Words

There is no “empty word” in TI LOGO. PRINT “ results in an error message, and BUTFIRST “A outputs a meaningless expression. To terminate a word procedure, you need to substitute a different condition for the statement

```
IF :WORD = " [OUTPUT. ..]
```

used in this book. Instead use

```
IF FIRST :WORD = :WORD THEN . . .
```

This condition will be true only for a one-letter word. For example, the procedure TRIANGLE becomes:

```

TO TRIANGLE :WORD
PRINT :WORD
IF FIRST :WORD = :WORD THEN STOP
TRIANGLE BUTFIRST :WORD
END

```

The *location* of the conditional line in the procedure also had to be changed to print the single-letter word before stopping.

### Section 5.4. Lists

Everything in this section can be carried out as is in TI LOGO if you use the command READLINE in place of READLIST.

### Section 5.5. Naming

Everything here is the same except that you must use “1 instead of 1 in this example near the beginning of section 5.5:

```
MAKE (WORD "PART "1) [DO MI SOL]
```

## Section 5.6. Conditional Expression and Predicates

Type as one line. See page 9.—

To express a complex conditional in TI LOGO, use THEN and ELSE, rather than two sets of brackets:

```
TO SIGN :N (Apple Logo)
IF :N < 0 [OUTPUT "NEGATIVE] [OUTPUT "POSITIVE]
END
```

```
TO SIGN :N (TI LOGO)
IF :N < 0 THEN OUTPUT "NEGATIVE
ELSE OUTPUT "POSITIVE
END
```

Another difference here is that the commands BOTH and EITHER must be substituted for the Apple Logo's AND and OR.

## Section 5.7. Details of Logo Syntax

There is a difference in the ways TI LOGO and Apple Logo parse arithmetical expressions contained in words and lists:

```
PRINT "3 + 4 (Apple Logo)
7
```

```
PRINT "3 + 4 (TI LOGO)
3 + 4
```

When the expression 3 + 4 is contained within a list, however, both versions of Logo do the same thing:

```
PRINT [3 + 4 5]
3 + 4 5 (a list of four elements)
```

When dealing with negative numbers, TI LOGO and Apple Logo parse expressions differently. In TI LOGO the expressions - 5 and - 5 are treated as equivalent:

```
PRINT - 5
- 5
```

```
PRINT - 5
- 5
```

However, in Apple Logo a space before the number indicates that - is being used as an operator:

```
PRINT - 5
- 5
PRINT - 5
NOT ENOUGH INPUTS TO -
```

The reverse effect happens when negative numbers are contained within lists:

```
PRINT [-1] (TI LOGO)
- 1 (a two element list)
PRINT [-1] (Apple Logo)
- 1 (a single element list)
```

This difference becomes critical if you want to extract a negative number from a list in TI LOGO:

```
PRINT FIRST [-1 2] (Apple Logo)
- 1
PRINT FIRST [-1 2] (TI LOGO)
-
```

TI Logo treats [-1 2] as a *three* element list, [-1 2]. If you want to extract a negative number from a list, you can get around this by using SENTENCE to type the number into the list in the first place:

```
MAKE "LIST SENTENCE (-1) 2
PRINT :LIST
- 1 2
PRINT FIRST :LIST
- 1
```

### Section 5.7.2. Using Parentheses

TI LOGO uses parentheses in the same way as Apple Logo. With the exception of SENTENCE, however, no TI LOGO commands can take multiple arguments; for example:

```
PRINT (WORD "A "B "C) (Apple Logo)
ABC
```

The same result can be accomplished by *nesting* repeated calls to WORD:

```
PRINT WORD "A WORD "B "C
ABC
```

The second WORD puts "B and "C together and the first WORD then puts "A and "BC together. Similar effects can be obtained with most of the commands which take multiple inputs in Apple Logo:

```
PRINT (OR (5 = 3) (5 = 4) (5 = 5)) (Apple)
TRUE
PRINT EITHER (5 = 3) EITHER (5 = 4)(5 = 5) (TI)
```

## CHAPTER 6

### Section 6.2. Random-Sentence Generators

Use RANDOM1 described above in place of RANDOM in the PICKRANDOM procedure. (However, you can use TI LOGO's RANDOM with no input in the second version of BABBLE. The output of RANDOM1 10 is the same as the output of RANDOM—a random number from zero to nine.)

### Section 6.3. NIM: a Game Playing Program

Use the REMAINDER procedure described above in SMARTMOVE.

## CHAPTER 7

### Section 7.2. Keyboard Input

PUTTILE can be used instead of SETCURSOR to print information at a particular location on the screen.

Use READLINE instead of READLIST. The READKEY procedure of Section 7.2.2 should also be changed:

```
TO READKEY
IF RC? OUTPUT READCHAR
OUTPUT []
END
```

### Section 7.3. The Dynaturtle Program

The dynaturtle program uses trigonometric functions which are not directly available in TI LOGO. They can be simulated with lists of numbers so that a program which approximates the correct dynamic motion can be developed. The procedure SET is required as part of STARTUP. A starting value of "FORCE must also be specified:

```
TO STARTUP
CLEARSCREEN
SET
MAKE "VX 0
MAKE "VY 0
MAKE "FORCE 3
END
```

Type as one line. See page 9.→

```
TO SET
MAKE "SIN (SENTENCE 0 1 2 3 2 1 0 (-1) (-2) (-3)
(-2) (-1))
MAKE "COS (SENTENCE 3 2 1 0 (-1) (-2) (-3) (-2)
(-1) 0 1 2)
END
```

Type as one line. See page 9.→

(The lists, "SIN and "COS must be typed using SENTENCE to avoid the problem with negative numbers described above.)



The other procedures change to:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R NEWRIGHT
IF :COM = "L NEWLEFT
IF :COM = "K KICK
END

TO KICK
MAKE "VX :VX + (:FORCE * SIN) / 3
MAKE "VY :VY + (:FORCE * COS) / 3
END

TO MOVETURTLE
SXY (XCOR + :VX) (YCOR + :VY)
END
```

and there are four new procedures required:

```
TO NEWRIGHT
RIGHT 30
MAKE "SIN SENTENCE (BUTFIRST :SIN) (FIRST :SIN)
MAKE "COS SENTENCE (BUTFIRST :COS) (FIRST :COS)
END

TO NEWLEFT
MAKE "SIN SENTENCE (LAST :SIN) (BUTLAST :SIN)
MAKE "COS SENTENCE (LAST :COS) (BUTLAST :COS)
END

TO SIN
OUTPUT FIRST :SIN
END

TO COS
OUTPUT FIRST :COS
END
```

The procedure DT remains the same.

#### Section 7.4. Input from the Paddles

### CHAPTER 8

#### Section 8.1. Reversing Words and Lists

TI LOGO has nothing comparable to paddle input at present.

Many activities in this chapter require modification due to differences in the way words and lists are handled in TI LOGO compared to versions of Logo for the Apple.

To reverse words, change the stop rule in REVERSE, because TI LOGO does not allow the empty word. Instead use:

```
IF :X = FIRST :X OUTPUT :X
```

In other words, if :X has only one letter, output that letter.

When we come to the palindrome project we have a serious problem, for TI LOGO does not allow word operations on numbers or arithmetical operations on words. We need procedures called MAKE.NUMBER and MAKE.WORD, that convert a word to a number and vice versa. With these procedures, the MAKE.PALINDROME procedure becomes:

```
TO MAKE.PALINDROME :NUMBER
PRINT :NUMBER
IF PALINDROME? MAKE.WORD :NUMBER THEN STOP
MAKE.PALINDROME :NUMBER +
 (MAKE.NUMBER REVERSE
 MAKE.WORD :NUMBER)
END
```

Type as one line. See page 9.—

In the second line, PALINDROME? needs a *word* as input, so :NUMBER must be converted. In the last line, + needs a *number* as input, but REVERSE needs a *word*. So, :NUMBER is converted to a word and reversed. This word is then converted back to a number so that the original :NUMBER can be added to its reverse.

This kind of activity sounds peculiar, but it works. In computer jargon it's called a "hack." People who dream up programs like this come to be called "hackers." Here are the procedures needed:

```
TO MAKE.NUMBER :WORD
IF FIRST :WORD = :WORD
 OUTPUT (CONVERT.ONE :WORD
 :WORDLIST
 :NUMLIST)
OUTPUT (10 * MAKE.NUMBER BUTLAST :WORD) +
 (CONVERT.ONE LAST:WORD)
 :WORDLIST
 :NUMLIST)
END
```

Type as one line. See page 9.—

Type as one line. See page 9.—

```
TO MAKE.WORD :NUMBER
IF :NUMBER / 10 = 0
 OUTPUT (CONVERT.ONE :NUMBER
 :NUMLIST
 :WORDLIST)
OUTPUT WORD (MAKE.WORD :NUMBER / 10)
 (CONVERT.ONE REMAINDER :NUMBER 10
 :NUMLIST
 :WORDLIST)
END
```

Type as one line. See page 9.—

Type as one line. See page 9.—

Type as one line. See page 9.—

```
TO CONVERT.ONE :ITEM :LIST1 :LIST2
IF :ITEM = FIRST :LIST1 OUTPUT FIRST :LIST2
OUTPUT CONVERT.ONE :ITEM
 BUTFIRST :LIST1
 BUTFIRST :LIST2
END
```

Before the process starts we need to define the two lists, using something like:

Type as one line. See page 9.—

```
TO SETUP
MAKE "NUMLIST (SENTENCE 0 1 2 3 4 5 6 7 8 9)
MAKE "WORDLIST (SENTENCE "0 "1 "2 "3 "4
 "5 "6 "7 "8 "9)
END
```

## Section 8.2. Procedures that Manipulate Lists

The only change here is in the procedure PICKRANDOM. Use the procedure RANDOM1 :NUMBER described above instead of RANDOM (section 8.2.2).

## Section 8.3. Radix Conversion

Here we again confront the problem of having to get around the fact that WORD does not work on numbers in TI LOGO. We can use the same conversion procedures described above. WORD needs a *word* as input, but BASE8 needs to output a number. This can be accomplished as follows:

Type as one line. See page 9.—

```
TO BASE8 :N
IF :N = 0 OUTPUT 0
OUTPUT MAKE.NUMBER
 (WORD (MAKE.WORD
 BASE8
 (QUOTIENT :N 8))
 (MAKE.WORD REMAINDER :N 8))
END
```

REMAINDER is the procedure described above. QUOTIENT is a TI LOGO command. Remember to use the SETUP command for word/number conversion before using BASE8. The reader can apply this method to the other procedures in the chapter, but the use of letters for numbers in bases larger than ten makes the process a lot trickier.

# CHAPTER 9

## Section 9.1. Hierarchical Structures

The Logo command LIST, which is not implemented in TI LOGO, is used to create hierarchical lists—lists of lists. Such lists can be typed in from the keyboard or created using FPUT and LPUT. Some of the programs described in this chapter can be implemented using the SENTENCE command.

The difference between LIST and SENTENCE is simple. LIST

makes lists of lists. SENTENCE strips away all list brackets and combines all elements into a *single* list:

```
PRINT LIST [A] [B]
[A] [B]
PRINT SENTENCE [A] [B]
A B
```

## Section 9.2. Programs as Data

The INSTANT program can be carried out with minor changes. Just substitute READLINE for READLIST in the procedures ASK and LEARN, and use the correct syntax for conditions in TI LOGO.

### Section 9.3.1. The DOCTOR Program

The DOCTOR program can be used with only the following minor changes. Use the RANDOM1 procedure in place of RANDOM and use READLINE instead of READLIST.

### Section 9.3.2. The ANIMAL Program

The ANIMAL program uses the LIST command to create hierarchical lists in the EXTEND.KNOWLEDGE and REPLACE procedures. LIST with *two* inputs can be written as a Logo procedure. This LIST procedure can be used with FPUT in EXTEND.KNOWLEDGE and REPLACE:

```
TO LIST :A :B
OUTPUT FPUT :A FPUT :B []
END
```

Type as one line. See page 9.—

```
TO EXTEND.KNOWLEDGE :NEW.QUESTION
:YES.ANSWER
:NO.ANSWER
```

Type as one line. See page 9.—

```
MAKE "KNOWLEDGE
REPLACE :KNOWLEDGE
:NO.ANSWER
(FPUT :NEW.QUESTION
(LIST :YES.ANSWER
:NO.ANSWER))
```

```
END
```

Type as one line. See page 9.—

```
TO REPLACE :TREE :NODE :REPLACEMENT
IF :TREE = :NODE OUTPUT :REPLACEMENT
IF WORD? :TREE OUTPUT :TREE
OUTPUT (FPUT QUESTION :TREE
(LIST REPLACE YES.BRANCH :TREE
:NO.ANSWER
:REPLACEMENT
REPLACE NO.BRANCH :TREE
:NO.ANSWER
:REPLACEMENT))
```

```
END
```

Only a few other minor changes are needed. Use READLINE instead of READLIST in the procedures ASK.YES.OR.NO and GET.SMARTER. Use the IF...THEN...ELSE syntax for conditions rather than brackets.

## CHAPTER 10

Commands that are the same in both versions of Logo have been omitted, and so have commands that are in TI LOGO but have no equivalent in Apple Logo.

### Section 10.1. Graphics Commands

|                              |                                                     |
|------------------------------|-----------------------------------------------------|
| BACKGROUND                   | TELL BACKGROUND                                     |
| CLEAN                        | COLOR                                               |
| DOT [20 100]                 | no equivalent                                       |
|                              | DOT (no input) – draws dot at the turtle's position |
| FENCE                        | no equivalent                                       |
| FULLSCREEN                   | no equivalent                                       |
| PEN                          | no equivalent                                       |
| PENCOLOR                     | COLOR                                               |
| POS (outputs list)           | XCOR and YCOR                                       |
| SETBG                        | COLORBACKGROUND                                     |
| SETH (SETHEADING)            | SH (SETHEADING)                                     |
| SETPC                        | SETCOLOR or SC                                      |
| SETPEN                       | no equivalent                                       |
| SETPOS (takes list as input) | SXY (takes two numbers as input)                    |
| SETX, SETY                   | SX, SY                                              |
| SHOWNP                       | no equivalent                                       |
| SPLITSCREEN                  | no equivalent                                       |
| TEXTSCREEN                   | no equivalent                                       |
| TOWARDS                      | no equivalent                                       |
| WINDOW                       | no equivalent                                       |
| WRAP                         | no equivalent                                       |

### Section 10.2. Numeric Operations

|                       |                          |
|-----------------------|--------------------------|
| ARCTAN                | no equivalent            |
| COS                   | no equivalent            |
| INT                   | no equivalent            |
| RANDOM (input number) | RANDOM (no input)        |
| REMAINDER             | implement as a procedure |
| RERANDOM              | no equivalent            |
| ROUND                 | no equivalent            |
| SIN                   | no equivalent            |
| SQRT                  | no equivalent            |

### Section 10.3. Word and List Operations

|       |                          |
|-------|--------------------------|
| COUNT | implement as a procedure |
| ITEM  | implement as a procedure |
| LIST  | implement as a procedure |

(Word operations in TI LOGO do not accept numbers as inputs.)

|                                                                   |                                                                                                                |                                                                                                                                                           |
|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Section 10.4. Defining and Editing Procedures</b>              | EDIT or ED (input names must be quoted or in a list)<br>EDNS<br>TO (enters procedure definition mode)          | EDIT (name does not have to be quoted. Will not accept a list, or ALL as input)<br>no equivalent<br>TO (enters edit mode)                                 |
| <b>Section 10.5. Conditional Expressions</b>                      | AND<br>OR<br>IFFALSE or IFF<br>IFTRUE or IFT                                                                   | BOTH<br>EITHER<br>IFF (short form only)<br>IFT (short form only)                                                                                          |
| <b>Section 10.6. Predicates Used With Conditional Expressions</b> | EMPTY<br>EQUALP<br>LISTP<br>MEMBERP<br>NUMBERP<br>WORDP                                                        | no equivalent<br>no equivalent<br>no equivalent<br>implement, as a procedure<br>NUMBER?<br>WORD?                                                          |
| <b>Section 10.7. Controlling Procedure Execution</b>              | No differences between the two versions.                                                                       |                                                                                                                                                           |
| <b>Section 10.8. Input and Output</b>                             | ASCII<br>BUTTONP<br>CHAR<br>CLEARTEXT<br>CURSOR<br>KEYP<br>PADDLE<br>.PRINTER<br>READLIST<br>SETCURSOR<br>SHOW | CHARNUM<br>no equivalent<br>PRINTCHAR<br>CLEARSCREEN<br>no equivalent<br>RC?<br>JOY<br>no equivalent<br>READLINE<br>no direct equivalent<br>no equivalent |
| <b>Section 10.9. Naming</b>                                       | DEFINEDP<br>LOCAL<br>NAME<br>NAMEP                                                                             | no equivalent<br>no equivalent<br>CALL<br>no equivalent                                                                                                   |
| <b>Section 10.10. Filing and Managing Workspace</b>               | BURY<br>CATALOG<br>ERALL<br>ERASEFILE<br>ERN                                                                   | no equivalent<br>use space bar while filing<br>no equivalent<br>no equivalent<br>no equivalent                                                            |

|         |               |
|---------|---------------|
| ERNS    | no equivalent |
| ERPS    | no equivalent |
| LOAD    | RECALL        |
| PACKAGE | no equivalent |
| PKGALL  | no equivalent |
| POALL   | PA            |
| PONS    | PN            |
| POPS    | no equivalent |
| POTS    | PP            |
| UNBURY  | no equivalent |

**Section 10.11. Debugging Aids**

No differences.

**Section 10.12. Editing Commands**

See TI Reference Manual for editing commands.

**Section 10.13. Other Control Characters**

See TI Reference Manual.

**Section 10.14. Primitives for Handling Properties**

No equivalent in TI LOGO.

**Section 10.15. Miscellaneous and Advanced Commands**

|            |               |
|------------|---------------|
| .BPT       | no equivalent |
| CATCH      | no equivalent |
| .CONTENTS  | CONTENTS      |
| COPYDEF    | no equivalent |
| .DEPOSIT   | no equivalent |
| DISK       | no equivalent |
| ERROR      | no equivalent |
| .EXAMINE   | no equivalent |
| LABEL      | no equivalent |
| NODES      | no equivalent |
| PRIMITIVEP | no equivalent |
| RECYCLE    | no equivalent |
| REPARSE    | no equivalent |
| SCRUNCH    | no equivalent |
| SETSCRUNCH | no equivalent |
| SETDISK    | no equivalent |
| THROW      | no equivalent |

**Section 10.16. Error Messages**

Only the most commonly encountered error messages will be compared.

|                                 |                            |
|---------------------------------|----------------------------|
| I DON'T KNOW HOW TO...          | TELL ME HOW TO...          |
| NOT ENOUGH INPUTS TO...         | TELL ME MORE               |
| I DON'T KNOW WHAT TO DO WITH... | TELL ME WHAT TO DO WITH... |
| NUMBER TOO BIG                  | no equivalent              |

CAN'T DIVIDE BY ZERO

TURTLE OUT OF BOUNDS

YOU TRIED TO DIVIDE BY  
ZERO

no equivalent





## References

---

1. Abelson, H. and diSessa, A. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge, MA, 1981.
2. Bowles, K. *Problem Solving Using Pascal*. Springer-Verlag, New York, 1977.
3. diSessa, A. "Unlearning Aristotelian Physics: A Study of Knowledge-Based Learning," *Cognitive Science*. (In press.)
4. Feurzeig, W., Papert, S., Bloom, M., Grant, R., and Solomon, C. "Programming Languages as a Conceptual Framework for Teaching Mathematics," *Report 1889*. Bolt, Beranek and Newman, Inc., November, 1969.
5. Feurzeig, W., Goldenberg, E.P., Lukas, G., Manis, V., Rubenstein, R., and Stachel, R. "The Logo-S Language and the Portable Logo System." Bolt, Beranek and Newman, Inc., 1980.
6. Goldberg, A.J., Robson, D., and Ingalls, D.H.H. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA. (In press.)
7. Goldenberg, P. *Special Technology for Special Children*. University Park Press, Baltimore, 1979.
8. Greenberg, B. "Notes on the Programming Language Lisp." MIT Student Information Processing Board, Cambridge, MA, 1978.
9. Howe, J.A.M., O'Shea, T., and Lane, F. "Teaching Mathematics through Logo Programming: An Evaluation Study." Department of Artificial Intelligence, University of Edinburgh, 1977.
10. Kay, A. "Microelectronics and the Personal Computer." *Scientific American* (September 1977).

11. Papert, S., and Solomon, C. "NIM: A Game-Playing Program." Memo 254, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1970.
12. Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.
13. Papert, S., diSessa, A., Watt, D., and Weir, S. "Final Report to the Brookline Logo Project: Assessment and Documentation of a Children's Computer Laboratory." Memos 53, 54, MIT Logo Project, Cambridge, MA, 1980.
14. Weir, S. "Logo and the Exceptional Child." *Kilobaud Microcomputing* 5, 9 (September 1981).
15. Weizenbaum, J. *Computer Power and Human Reason*. W.A. Freeman & Co., San Francisco, 1976.
16. Winston, P., and Horn, B. *Lisp*. Addison-Wesley, Reading, MA, 1981.

# Index

---

- Abbreviations, 8
- Absolute value, 80
- Activation, 38, 40
- Addition, 75
- AND, 93, 183
- ANIMAL program, 162
- Arcs, 29
- ARCTAN, 177
- Arctangent, 177
- Arithmetic, 75, 177
- Arrow keys, 4, 5, 191
- ASCII, 185
- Association list, 147
  
- BACK, 4, 173
- BACKGROUND, 19, 173
- BF (*see* BUTFIRST)
- Binary tree, 40
- BK (*see* BACK)
- BL (*see* BUTLAST)
- Body, 11
- Bowles, K., xi
- Brackets, 1, 9, 83
- BURY, 68, 188
- BUTFIRST, 81, 179
- BUTLAST, 81, 179
- BUTTONP, 124, 185
  
- Cartesian coordinates, 18, 123, 176, 177
- CATALOG, 70, 188
- CATCH, 193
- CHAR, 185
- Character input, 117, 186
- Circles, 29
- CLEAN, 18, 173
- CLEARSCREEN, 3-5, 173
- CLEARTEXT, 116, 185
- CO (*see* CONTINUE)
- Color, 18
- Conditional, 36, 91, 183, 184
- CONTINUE, 73, 190
- Control characters, 1, 12, 191, 192
- COPYDEF, 194
- COS, 76, 178
- Cosine, 76, 178
- COUNT, 132, 180
- CS (*see* CLEARSCREEN)
- Cursor, 5, 116, 186
- CURSOR, 116, 186
- CTRL key, 1
- CTRL-A, 5, 191
- CTRL-B, 5, 12, 191
- CTRL-C, 13, 21, 69, 191
- CTRL-D, 6, 12, 191
- CTRL-E, 6, 191
- CTRL-F, 191
- CTRL-G, 9, 13, 21, 33, 191, 192
- CTRL-H, 191
- CTRL-K, 6, 191
- CTRL-L, 16, 21, 191, 192
- CTRL-M, 191
- CTRL-N, 13, 191
- CTRL-O, 13, 191
- CTRL-P, 13, 191
- CTRL-Q, 192
- CTRL-S, 21, 192
- CTRL-T, 21, 192
- CTRL-U, 192
- CTRL-V, 16, 192
- CTRL-W, 193
- CTRL-Y, 6, 192
- CTRL-Z, 73, 193
  
- Debugging, 72, 190
- DEFINE, 152, 181
- DEFINEDP, 187
- diSessa, Andy, xi, xii, 119
- DISK, 194
- Division, 75
- DOCTOR program, 158
- DOT, 173
- Dots, 24
- Draw mode, 21
- Drescher, Gary, xi
- Dynaturtle, 119

- ED (*see* EDIT)
- EDIT, 12, 18, 69, 182
- Edit buffer, 69
- Edit mode, 21
- Editing commands, summary, 190
- Editor, 5, 12
- EDNS, 182
- Empty list, 84, 184
- Empty word, 82, 184
- EMPTY, 184
- END, 11, 182
- EQUALP, 184
- ER (*see* ERASE)
- ERALL, 68, 188
- ERASE, 68, 188
- ERASEFILE, 70, 188
- ERN, 188
- ERNS, 68, 189
- ERPS, 68, 189
- ERROR, 194
- Error messages, 7, 17, 196
- Errors, 196
- Errors, typing, 5
- ESC key, 1, 190
- ESC V, 16, 192
- ESC- $\leftarrow$ , 192
- ESC- $\rightarrow$ , 192
- Exponential notation, 76
  
- FALSE, 92
- FD (*see* FORWARD)
- Feurzeig, W., xi
- FENCE, 20, 174
- Files, 69
- Filing, 188
- FIRST, 81, 84, 144, 180
- FOR, 157
- FORWARD, 4, 174
- FPUT, 145, 180
- Free variables, 89
- FULLSCREEN, 21, 174
- Fullscreen mode, 21
  
- Gargarian, Gregg, xi
- Global variables, 89
- GO, 194
- Goldberg, A., x, xi
- GPROP, 149, 193
- Graphics commands, 173
- Gross, Mark, xi
  
- Hain, Stephen, xi
- Hard copy, 72
- Hardebeck, Edward, xi
  
- HEADING, 174
- HIDETURTLE, 4, 174
- Hierarchical structure, 142
- HOME, 18, 174
- Howe, J., xi
- HT (*see* HIDETURTLE)
  
- IF, 36, 91, 183
- IFFALSE, 91, 183
- IFF (*see* IFFALSE)
- IFTRUE, 91, 183
- IFT (*see* IFTRUE)
- Infix operators, 96
- Input, 4, 23
- INSTANT, 153
- INT, 77, 178
- Integers, 77
- ITEM, 132, 180
  
- Kay, A., xi
- Keyboard, 1
- KEYP, 118, 186
- Klotz, Leigh, xi
  
- LABEL, 195
- LAST, 81, 84, 144, 180
- LEFT, 4, 174
- LENGTH, 132
- Level, 73
- LIST, 146, 180
- List operations, 179
- LISTP, 184
- Lists, x, 9, 83, 143
- LOAD, 70, 189
- LOCAL, 187
- Local variables, 26, 88
- LOOKUP, 147
- LPUT, 145, 181
- LT (*see* LEFT)
  
- MAKE, 86, 187
- Mantissa, 77
- MEMBER?, 135
- MEMPERP, 132, 185
- Modes, 20
- Multiplication, 75
  
- NAME, 187
- NAMEP, 188
- Names, 26, 86
- Nim, 103
- NODES, 195
- Nodraw mode, 20

NOT, 93, 183  
 Number, 77  
 NUMBERP, 100, 185  
 Numeric operations, 75, 177

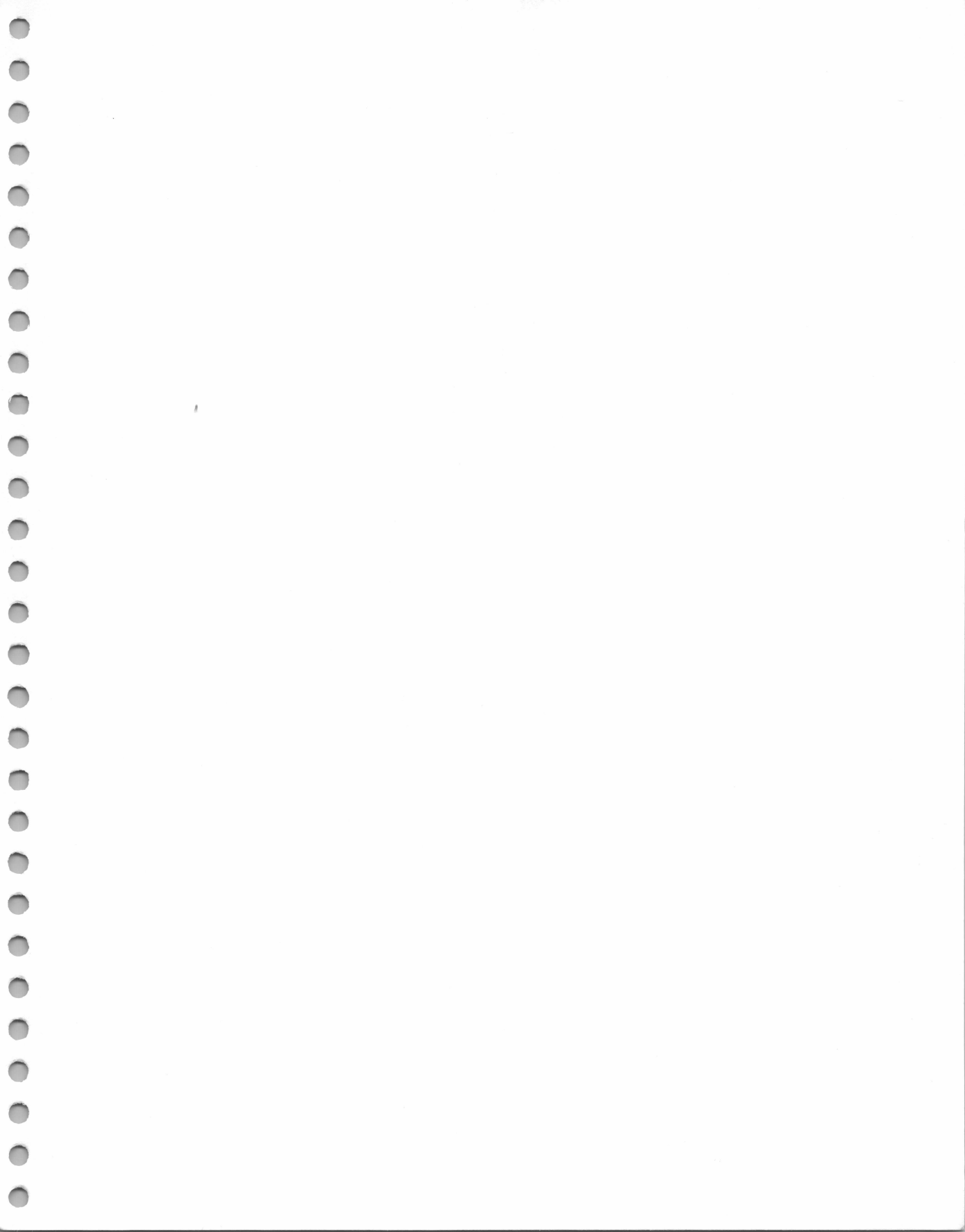
OP (*see* OUTPUT)  
 OR, 93, 183  
 OUTPUT, 79, 185

PACKAGE, 68, 189  
 PADDLE, 123, 186  
 Palindrome, 131  
 Papert, Seymour, xi, 28, 104  
 Parentheses, 96  
 PAUSE, 72, 190  
 PC (*see* PENCOLOR)  
 PD (*see* PENDOWN)  
 PE (*see* PENERASE)  
 PEN, 174  
 PENCOLOR, 18, 174  
 PENDOWN, 4, 175  
 PENERASE, 19, 175  
 PENREVERSE, 19, 175  
 PENUP, 4, 175  
 Physics, 119  
 PICK, 132, 134  
 PICKRANDOM, 101  
 Pig Latin, 136  
 PKGALL, 189  
 PLIST, 149, 193  
 PO, 14, 67, 189  
 POALL, 68, 189  
 POLY, 33  
 PONS, 68, 189  
 POPS, 68, 190  
 POS, 175  
 POTS, 15, 67, 190  
 PPROP, 149, 193  
 PPS, 193  
 PR (*see* PRINT)  
 Predicate, 36, 91, 184  
 Prefix operators, 96  
 Primitive, ix, 10  
 PRIMITIVEP, 195  
 PRINT, 2, 97, 115, 186  
 Printers, 72  
 Printout, 14, 67  
 Private library, 26, 38  
 Procedure, 10, 79  
 Procedure editor, 12  
 Procedure, body, 11  
 Procedure, title, 10  
 PRODUCT, 178  
 Prompt, 2, 73  
 PU (*see* PENUP)  
 PX (*see* PENERVERSE)

Quiz program, 99  
 QUOTIENT, 77, 178

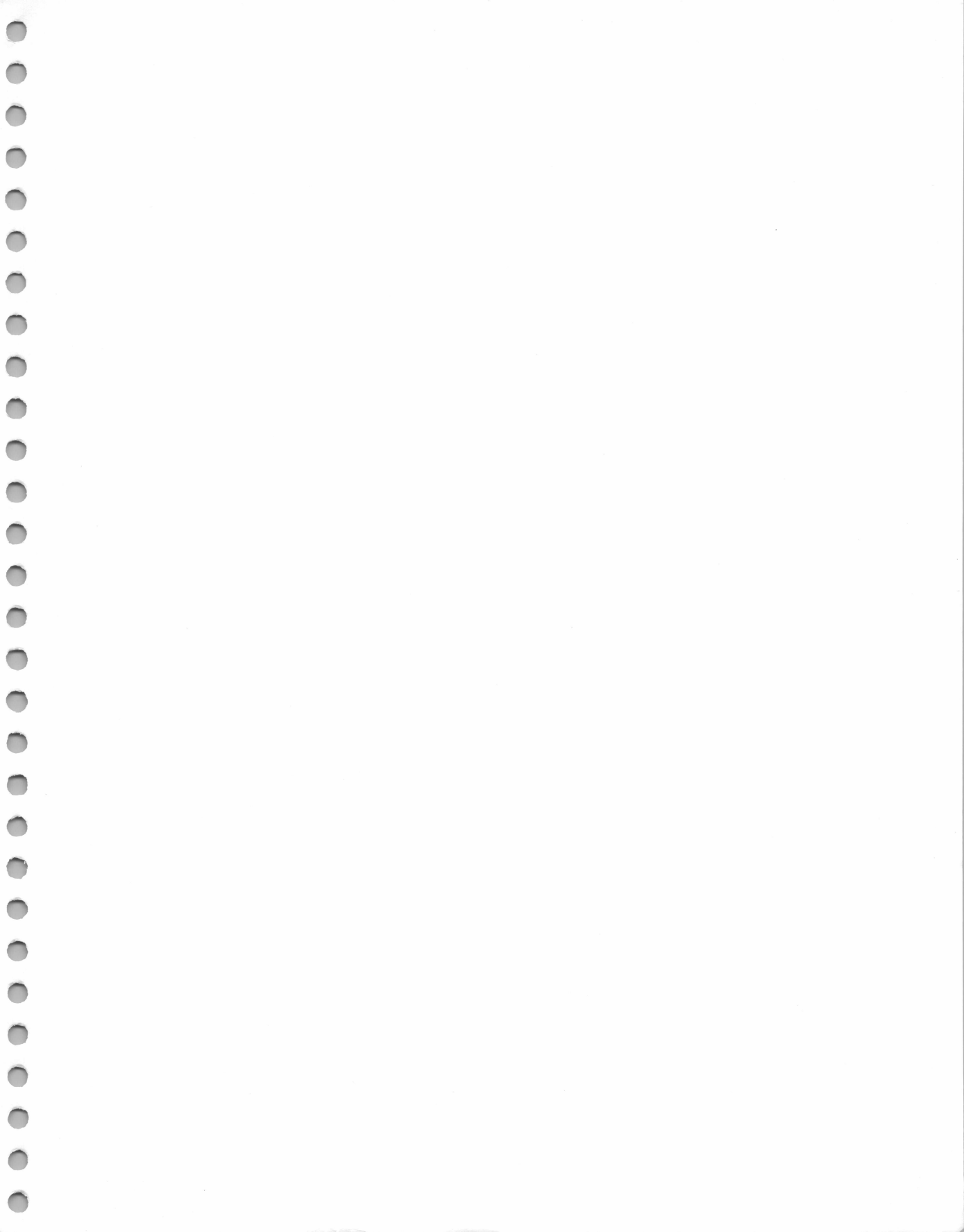
Radix conversion, 137  
 RANDOM, 78, 178  
 RC (*see* READCHAR)  
 READCHAR, 117, 186  
 READLIST, 88, 186  
 READNUMBER, 100  
 Real numbers, 77  
 Recalling lines, 6  
 Recursion, 32, 37, 40, 125  
 Recursive designs, 42  
 RECYCLE, 195  
 Reduction step, 129  
 REMAINDER, 78, 178  
 REMPROP, 149, 193  
 REPARSE, 195  
 REPEAT command, 8, 185  
 REPT key, 1, 5, 191  
 RERANDOM, 78, 178  
 RESET key, 1  
 RETURN key, 1, 2  
 REVERSE, 125  
 RIGHT, 4, 175  
 RL (*see* READLIST)  
 ROUND, 77, 179  
 RT (*see* RIGHT)  
 RUN, 149, 185  
  
 SAVE, 70, 190  
 SCRUNCH, 195  
 SE (*see* SENTENCE)  
 SENTENCE, 84, 97, 143, 145, 181  
 Sentence generator, 101  
 SETBG, 18, 175  
 SETCURSOR, 116, 186  
 SETDISK, 196  
 SETH (*see* SETHEADING)  
 SETHEADING, 18, 175  
 SETPC, 18, 175  
 SETPEN, 176  
 SETPOS, 18, 119, 123, 176  
 SETSCRUNCH, 195  
 SETX, 176  
 SETY, 176  
 SHOW, 186  
 SHOWNP, 176  
 SHOWTURTLE, 5, 176  
 SIN, 76, 179  
 Sine, 76, 179  
 Sobalvarro, Patrick, xi  
 Solomon, Cynthia, xii, 103  
 Spaces in Logo lines, 94  
 SPLITSCREEN, 21, 176  
 Splitscreen mode, 21

- SQRT, 76, 179
- ST (*see* SHOWTURTLE)
- STOP, 36, 185
- Stop rule, 41, 130
- Subprocedure, 12
- SUBST, 148
- Subtraction, 75
- SUM, 179
- Syntax, 94
- Tail recursion, 37
- TEST, 91, 183
- TEXT, 156, 182
- TEXTSCREEN, 21, 176
- THING, 88, 188
- THROW, 73, 147, 196
- TI LOGO, 201
- Title, 10
- Title line, 10, 15
- TO, 10, 182
- TOPLEVEL, 73, 147
- TOWARDS, 18, 176
- Tree, 40
- Tree structure, 142
- TRUE, 92
- Turtle, x, 3, 4,
- TYPE, 187
- UNBURY, 69, 190
- WAIT, 187
- Watt, Dan, xi, xii, 119
- Weizenbaum, J., 158
- WHILE, 156
- WINDOW, 20, 176
- WORD, 82, 181
- WORDP, 185
- Word operations, 179
- Workspace, 67, 188
- WRAP, 20, 176
- Wraparound, 20, 176
- XCOR, 18, 177
- YCOR, 18, 177
- \* 75, 177
- + 75, 177
- 75, 177
- .BPT, 193
- .CONTENTS, 194
- .DEPOSIT, 194
- .EXAMINE, 194
- .PRINTER, 72, 186
- / 75, 177
- : 87
- < 36, 184
- = 36, 184
- > 36, 184















The name Logo describes not only the evolving family of computer languages detailed in these pages, but also a philosophy of education that makes full and innovative use of the teaching potential of modern computers.

Readers of this book will see that the designers' vision of Logo as a virtually unlimited educational tool has now become a reality. Logo enables even young children to use the computer in self-directed ways rather than merely responding to it. Moreover, this same Logo is also an easy-to-use general programming system of considerable power and expression.

**Apple® Logo** teaches programming techniques through Turtle Geometry—a series of exercises involving both Logo programming and geometric concepts. Later chapters illustrate more advanced projects, such as the famous DOCTOR program with its simulated psychotherapist and an INSTANT program that enables parents and teachers to create a programming environment for preschool children.

**Apple® Logo** is directed toward the Apple II user, but it also includes a version of Logo for users of the Texas Instruments 99/4 computer.

Harold Abelson is Associate Professor of Electrical Engineering and Computer Science and heads the Educational Computing Group at M.I.T., where he received his doctorate in 1973. His other books include *Calculus of Elementary Functions* (1970) and *Turtle Geometry* (1981), and he is now writing an introductory computer science text.